BIBLIOTECA BASICA

INFORMATICA

PROGRAMANDO COMO ES DEBIDO

9

algoritmos y otros elementos necesarios



INGELEK

BIBLIOTECA BASICA INFORMATICA

PROGRAMANDO COMO ES DEBIDO

9

algoritmos
y otros elementos
necesarios

INGELEK

Director editor:
Antonio M. Ferrer Abelló.

Director de producción:
Vicente Robles.

Coordinador y supervisión técnica:
Enrique Monsalve.

Colaboradores:
Angel Segado.
Patricia Mordini.
Margarita Caffaratto.
Marina Caffaratto.
Francisco Ruiz.
Jorge Juan Monsalve.
Beatriz Tercero.
Fernando Ruiz.
Casimiro Zaragoza.

Diseño:
Bravo/Lofish.

Dibujos:
José Ochoa.

ISBN del tomo: 84-85831-40-3
ISBN de la obra: 84-85831-31-4
Fotocomposición: Pérez Díaz, S. A.
Imprime: Héroes, S. A.
Depósito Legal: M-37.688-1985

INDICE

PROLOGO

En este volumen afrontaremos el delicadísimo problema de la programación. El tema está desarrollado basándose especialmente en el BASIC, aunque tampoco faltarán esporádicas referencias al Pascal, lenguaje del que ya conocemos algo, gracias al número anterior de la BBI (Introducción al Pascal).

Los temas que se podrían tratar, con el riesgo de repetir lo dicho y repetido en otros textos y en las revistas especializadas, son muchísimos: desde el algoritmo de resolución más adecuado, hasta la mejor utilización de la máquina de que se dispone, pasando por el estilo del programa, visto especialmente bajo el perfil de sus características "user friendly" (manejo claro y sencillo). En efecto, pasaron ya los días en los que un programa servía sólo para el uso de quienes lo habían redactado y que, por lo tanto, en caso de problemas sólo podían enfadarse con ellos mismos (y ya sabemos que con nosotros mismos tendemos a ser especialmente indulgentes).

Entre tantas posibilidades nos hemos decidido a concretar al máximo la problemática de los algoritmos resolutivos (aunque sin ninguna pretensión de desarrollar un tratado exhaustivo). Hemos procedido a través de ejemplos de creciente complejidad y limitándonos a un BASIC muy estándar. Así pretendemos alcanzar dos objetivos:

- facilitar la comprensión de la problemática que supone la realización y puesta en marcha de un programa;
- salir de los límites de un BASIC demasiado vinculado a las interioridades del propio microsistema.

Pensamos sin falsa modestia que este tratamiento será apreciado por todos los que quieran entender lo que quiere decir PROGRAMAR.

Los dos últimos capítulos incluyen problemas muy prácticos que buscan hacer todo lo veloz y eficiente que se pueda el "caracol" que es el BASIC interpretado. Por razones de espacio y de concreción nos referimos al popular Commodore 64, pero los temas y problemas tocados resultan muy típicos, por ejemplo, el problema del "garbage collection" (identificación de la información inservible) de las cadenas. Es por esto que los consejos que damos hay que considerarlos prácticamente de aplicación universal.

CAPITULO I

EL GALIMATIAS DE LOS ALGORITMOS

¿Es el software un arte?

Esta pregunta, realmente crucial, afecta hoy en día no sólo a los usuarios, sino también a una gran cantidad de empresas interesadas en transformar esta actividad artesanal en industrial. "The art of programming": así se titula el talismán para el programador, una monumental enciclopedia del Software escrita por Knuth, uno de los "genios" en esta materia. En sus muchos tomos el autor pasa lista, explicándolos y clasificándolos, a la mayor parte de los tipos de algoritmos usados en la programación. Muchas veces ocurre que principiantes y semiexpertos copian un procedimiento de una revista o un libro, que, a su vez, está recogido de esta fuente sagrada del software.

No hay que escandalizarse por ello. Además, aunque esto sirva desde un punto de vista general. Es muy difícil, cuando se tiene un problema particular, encontrar una receta adecuada precisamente para nuestro caso. Quizá exista en algún sitio (casi está ya todo inventado en este mundo), pero ¿dónde? En estos casos no queda más remedio que arreglárselas solos. Por desgracia, a menudo surge la inquietante pregunta: sí, estoy dispuesto, ¿por dónde empiezo?

En la programación, efectivamente, hay ocasiones en las que la la flexibilidad y la libertad de acción se traducen en tragedia, peor incluso que la del célebre asno que, puesto delante de dos montones de heno, se murió de hambre por no saber por cuál decidirse. En el caso del software, más que de dilema se debería hablar de multidilemas (dilemas super ramificados); las reglas (sen-

tencias, bucles, GOTO) en realidad son pocas, pero ¿cómo se pueden combinar adecuadamente? Que nosotros sepamos, hasta ahora nadie ha conseguido hacer otra cosa que no sea proporcionar ejemplos, recomendando que, en casos semejantes, se proceda "por analogía". Sustancialmente, en este libro haremos lo mismo (si dijéramos lo contrario no seríamos honrados).

En definitiva, ¿quiere todo esto decir que el software es un arte? Según nuestro punto de vista sí, a pesar de los desesperados esfuerzos para producirlo de manera, incluso, "automática", mediante los más dispares métodos. Tomemos como ejemplo el caso del Last One, programa que genera un listado BASIC basándose tan sólo en las especificaciones del problema dadas por el usuario. Después de semanas enteras aprendiendo su uso y empleándolo, ¿cuántos son los que se han dado cuenta que la cosa marcha siempre en aquellos casos en los que se las habrían arreglado perfectamente solos, por normales que fueran sus conocimientos, con menos trajines y obteniendo programas más eficientes, veloces y compactos? En los casos inéditos o, lo que es lo mismo, en nuestros problemas reales, estamos otra vez como al principio: la pantalla (o la impresora) permanecen inactivas: necesitan una idea, lo mismo que nos ocurre a nosotros.

Como muchos ya saben, la idea de partida, ésa que puede llegar mediante la intuición creativa, por obra y gracia de la meditación trascendental, o por cualquier otro medio por raro que pueda parecernos, se llama "algoritmo", término que deriva del nombre del matemático árabe Al-Khuwarizmi. Un algoritmo es un conjunto de reglas o formas de actuar para la resolución de un problema.

La figura 1 muestra el esquema general de la génesis y realización de un programa. Los especialistas ofrecen infinidad de variantes de este esquema, pero en todas ellas hay un punto común: el comienzo es BUSCAR UN ALGORITMO RESOLUTORIO, y éste es el punto más delicado de todo el proceso.

Sin embargo, no queremos desanimar a los que se acercan por primera vez a un ordenador. El software es un arte, pero también se puede aprender; para las personas con fuerza de voluntad se trata de un desafío excitante y estimulante.

Algoritmo euclidiano y otros algoritmos numéricos

Dado que no parece posible elaborar una teoría desde la cual "deducir" —como ocurre con los sistemas de ecuaciones lineales y con el Algebra de Boole— los casos particulares, lo más sencillo es dejarnos de discursos y entrar de lleno en los ejemplos que nos servirán más o menos como medio de establecer analogías.

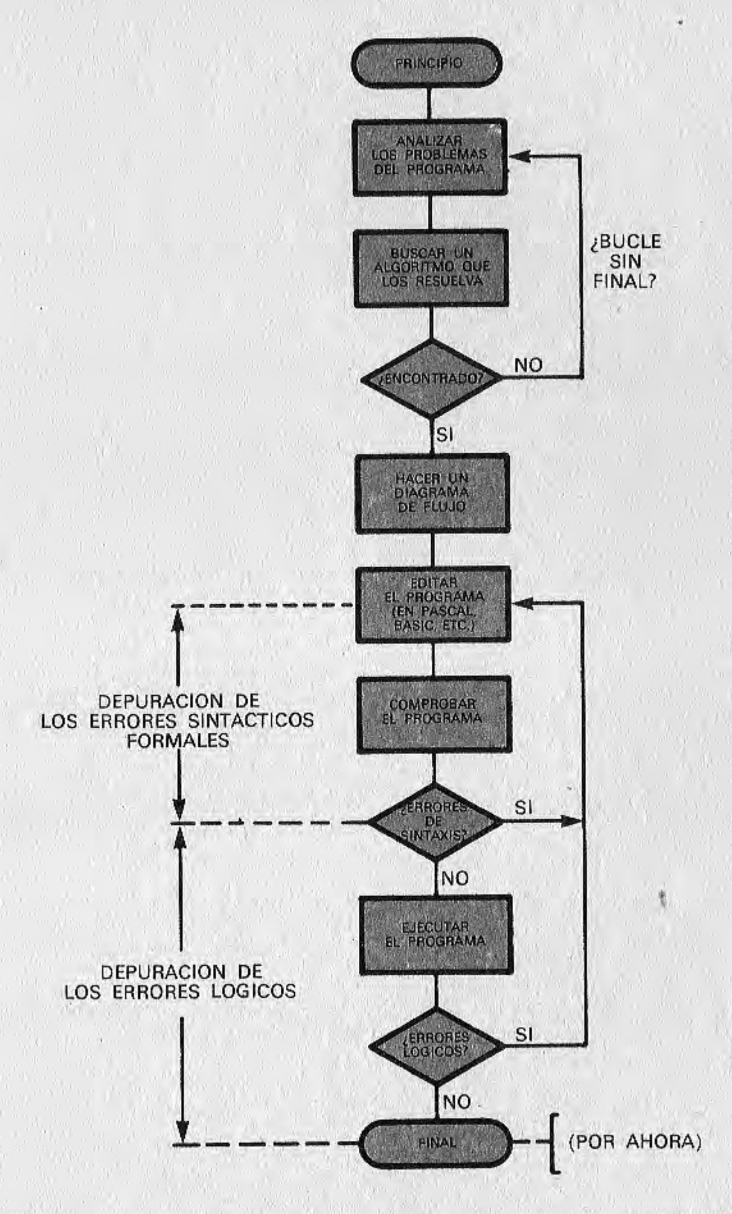


Figura 1.—El clásico ciclo de producción del software (simplificado) subdividido en las fases de implementación y debugging (pruebas y corrección de errores). Para problemas desconocidos las dificultades mayores surgen al principio: imaginar un algoritmo es una recomendación demasiado vaga. Son necesarias intuición y experiencia.

Empecemos por los más simples y clásicos, partiendo del que quizá es el más antiguo de todos: el algoritmo de Euclides. Sirve para hallar el M.C.D. (Máximo Común Divisor) de dos números naturales, es decir, enteros positivos. Sean estos X e Y. El algoritmo euclidiano se define de esta forma:

paso 1	si X e Y son iguales, ha terminado: el M.C.D. es el valor común.
paso 2	reste al mayor el menor.
paso 3	sustituya el mayor por el resultado anterior.
paso 4	vuelva otra vez al paso 1.

El procedimiento descrito, si se reflexiona bien, es delicioso. Muchos de nosotros habríamos resuelto el problema descomponiendo el número en factores primos y eligiendo los comunes con el mínimo exponente, como nos han enseñado en el colegio. Hay que tener en cuenta que Euclides, al igual que Al-Khuwarizmi no sabía nada de ordenadores, y en cambio supo:

- idear un proceso por aproximaciones sucesivas o, como se suele decir en Informática," iterativo".
- introducir la idea que en los lenguajes de programación está expresada por sentencias del tipo X=X-Y (donde el signo = debe entenderse como "convertirse en").

En la figura 2 se representa el correspondiente diagrama de flujo; el programa en BASIC correspondiente sería:

```
 5 DEFINT X,Y,A,B
10 INPUT X,Y:A=X:B=Y
20 IF A=B THEN 50
30 IF A>B THEN A=A-B:GOTO 20
40 B=B-A:GOTO 20
50 PRINT "EL M.C.D. DE ";X;" E ";Y
60 PRINT "ES IGUAL A ";A:END
```

(NOTA: en los dialectos BASIC en los que no exista la instrucción DEFINT para la definición explícita de variables enteras, se usarán variables tipo X%, Y%, o bien nos arreglaremos con las reales).

La comprobación empírica del algoritmo no es difícil. Por ejemplo, con la pareja X = 30; Y = 12 los sucesivos valores de A y B elaborados por el programa son:

A	B
30	12
18	12
6	12
6	6

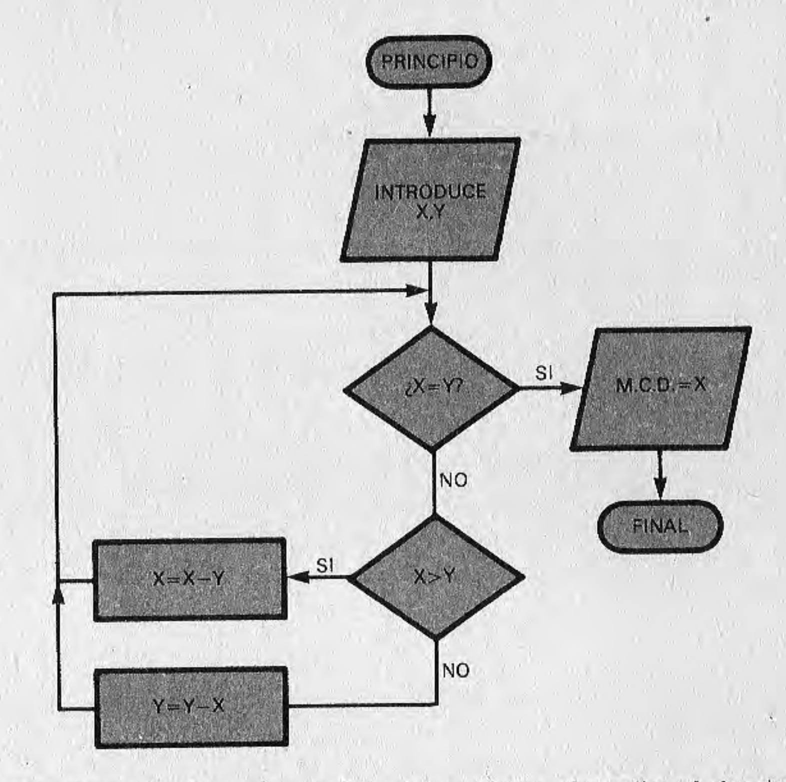


Figura 2.—Diagrama de flujo del programa que aplica el algoritmo de Euclides para obtener el M.C.D.

En cuanto a la demostración, esta tiene que ver con que el M.C.D. de dos números naturales A y B, con A>B, es el mismo que el de A-B y B. Se pueden encontrar analogías con el cálculo del cociente entero, resultado de dividir dos números mediante la sustracción repetida del divisor DSOR del dividendo DDO:

```
 5 DEFINT DSOR,DDO
10 INPUT DSOR,DDO
20 R=DSOR:Q=0:REM Q=COCIENTE:R=RESTO
30 IF R>DDO THEN 50
40 R=R-DDO:Q=Q+1:GOTO 30
50 PRINT R,Q:END
```

Naturalmente, la división entera la utilizamos aprovechando el hecho de que los ordenadores la "saben hacer", y así las 4 últimas líneas anteriores se reducen a una sola:

```
10 Q=INT(DDD/DSOR):R=DDD-Q*DSOR
```

También el algoritmo de Euclides puede hacerse más rápido por medio de divisiones sucesivas. Este asunto, sin embargo, lo dejamos como ejercicio al igual que (para los más avanzados) la demostración de cómo funciona, añadiendo dos variables auxiliares "s" y "t", el siguiente (sub)programa. Para que haya más variedad está escrito en Pascal y sirve para calcular el m.c.m. (mínimo común múltiplo):

```
PROCEDURE mincomult (x,y:INTEGER);VAR m.c.m:INTEGER);
VAR s,t:INTEGER;
 s:=x;t:=y;
 WHILE x<>y DO
  IF x>y THEN BEGIN x:=x-y;s:=s+t END;
     ELSE BEGIN y:=y-x;t:=t+s END;
 m.c.m.:=(s+t) DIV 2 (* DIV es la division entera *);
 END;
```

Los que lo intentaron podrán ahora comprobar su solución al problema de calcular el M.C.D. mediante divisiones enteras repetidas. Aquí está:

```
10 INPUT X,Y:A=X:B=Y
20 IF A<B THEN C=A:A=B:B=C
30 A=A-INT(A/B)*B
40 IF A<>0 THEN 20
50 PRINT "EL M.C.M DE ";X;" Y ";Y
60 PRINT "ES: ";B
```

Como se ve, consiste simplemente en obtener repetidamente el resto de la división entera entre A y B, haciendo que sustituya al mayor. Vea como en la línea 20 el intercambio entre A y B (a través de la variable de servicio C) permite que en A siempre se tenga el mayor de la pareja. Al final, el M.C.D. se encuentra en B, por ejemplo:

A	B
18	3
0	

M.C.D.= 3

A	B
30	12
6	12
12	6
0	

M.C.D. = 6

Por lo tanto hemos conseguido revisar a Euclides y, al mismo tiempo, hemos proporcionado una variante de su célebre algoritmo.

Antes de continuar con otros ejemplos, nos parece oportuno introducir una importante reflexión general (no lo llamaremos "principio", pues sería demasiado enfático).

PRIMERA REFLEXION: el desarrollo de un determinado algoritmo (y su correspondiente programa) tiene, a menudo, un carácter evolutivo.

En otras palabras: no hay una única solución; estudiando y probando se puede encontrar alguna variante, quizá mejor. Algunas veces la nueva solución favorece un determinado factor, por ejemplo, la velocidad, en perjuicio de algún otro, por ejemplo, la claridad o lo compacto del programa. Lo más importante, naturalmente, es que funcione.

Para ceñirnos al tema vamos a proponer un ejercicio muy fácil: encuentre un algoritmo para escribir la tabla de los cuadrados de los números naturales, utilizando solamente la suma (la multiplicación y la elevación a potencia están por tanto prohibidas, sino sería demasiado banal). La solución, por si no la encuentran, se la decimos nosotros:

```
10 IMPAR=1:NC=1
20 FOR N=0 TO 100
30 PRINT N,NC
40 IMPAR=IMPAR+2:NC=NC+IMPAR
```

Para entender algo bastará con dar un vistazo a la tabla siguiente:

N	IMPAR	NC
0	1	1
1	3	4
2	5	9
3	7	16

Comprenderá en seguida que el programa trabaja poniendo al día la serie de números impares en la variable IMPAR, y aña-

diendo el resultado al valor anterior de NC. Así, 3 + 1 es 4; 5 + 4 es 9, etc. La demostración se basa en el desarrollo del binomio de Newton.

$$(N + 1)^2 = N^2 + 2N + 1.$$

relación que se lee así: el cuadrado del elemento siguiente a N, es decir, N+1 (primer miembro), es igual al cuadrado de N más el número impar (2N+1).

Para elaborar la tabla de los cuadrados y cubos, sólo con la suma se puede partir de la siguiente fórmula:

$$(N+1)^3 = N^3+3N^2+3N+1$$

teniendo en cuenta que 3N= N+N+N.

En la figura 3 se ha reproducido el diagrama de flujo del programa BASIC que sigue:

```
 5 INPUT MAX:CP=2
10 FOR NP=3 TO MAX STEP 2
20 LIM=SQR(NP)+1
30 TEST=3
40 IF TEST>LIM THEN 70
50 IF NP=INT(NP/TEST)*TEST THEN 80
60 TEST=TEST+2:GOTO 40
70 PRINT NP:CP=CP+1
80 NEXT NP
100 PRINT:PRINT "ENTRE 1 Y ";MAX;" HAY ";
110 PRINT CP;" NUMEROS PRIMOS
```

Se trata de un programa que genera e imprime los números primos siguientes a 1 y 2. Una vez establecido en la línea 5 el valor "MAX" al cual se quiere llegar, comienza el bucle (líneas 10-80) que recorre la serie de los números impares: 3, 5, 7, 9... Cada vez (línea 50) se realiza la prueba de divisibilidad. ¿Con qué? No con los números primos anteriores, sino con la serie de los números impares, desde 3 en adelante, serie más amplia y que comprende la de los números primos (todo número primo debe ser impar). Con este fin, de la línea 30 sale otro bucle (quién lo prefiera puede realizarlo también con FOR... NEXT), que genera sucesivamente los números impares en la variable TEST; si la prueba de divisibilidad entre NP y TEST tiene éxito, se salta al próximo NP (línea 80), pues el actual no vale, si no, se vuelve a probar con TEST+2.

Algunos pensarán que el problema parece bastante sencillo,

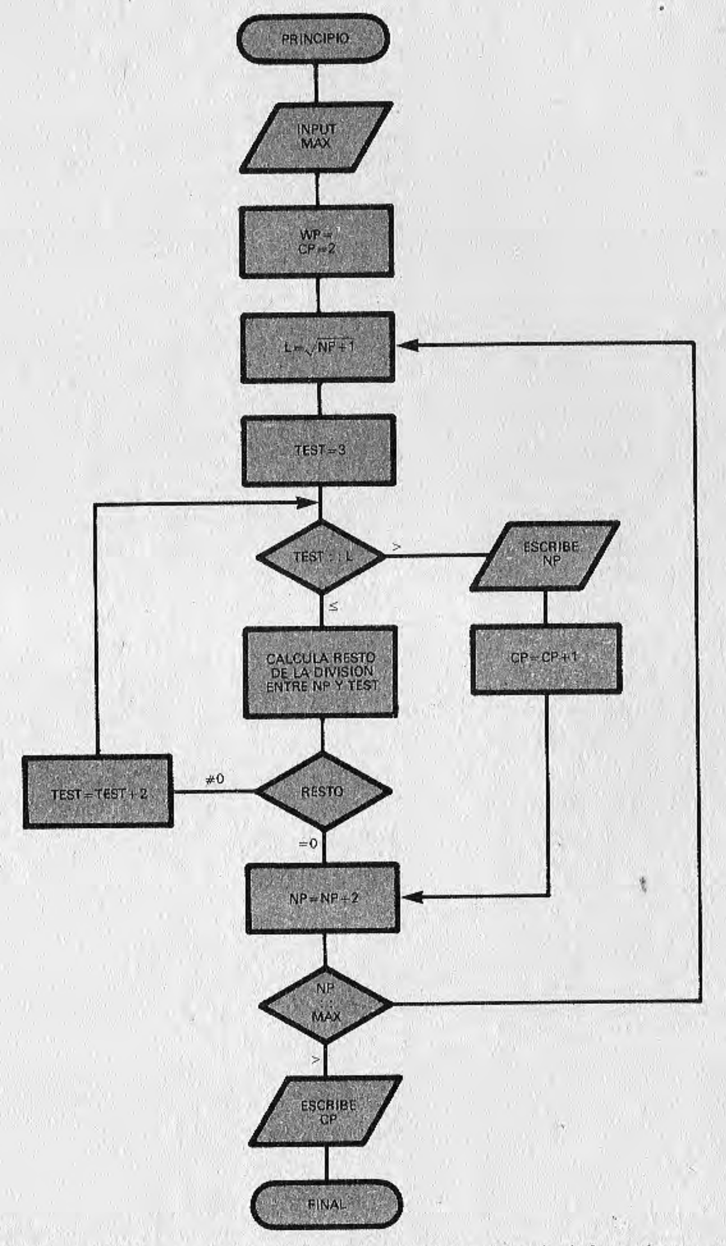


Figura 3.—Procedimiento para encontrar e imprimir los números primos presentes entre 3 y MAX. El TEST de divisibilidad de cada nuevo NP se hace con la serie entera de números impares desde 3.

pero ¿entienden la línea 20? En ella se calcula el valor de LIM, bajo el que se hace variar TEST, como la raíz cuadrada (por exceso) del NP en curso. En efecto, se demuestra que es superfluo seguir adelante con la prueba de divisibilidad si TEST supera LIM (se hace un salto a la línea 70); entonces NP se proclama primo y, por lo tanto, se incrementa el cuenta-primos CP (hay que notar que al principio CP se pone para contar el 1 y el 2).

El hecho de basar la prueba de divisibilidad en la raíz cuadrada (si 123 ha resultado indivisible por todos los impares desde 3 a 13, es inútil continuar con 15, 17, etc), nos lleva a nuestra "primera reflexión"; pues la primera idea sería hacer coincidir LIM con NP; sin embargo, al ponerlo como en el ejemplo, mejora notablemente la velocidad.

Aunque el procedimiento precedente pueda resultar obligado, por ejemplo, en las calculadoras de bolsillo (algunas tienen el BASIC pero carecen de memoria, y, a veces, también de la posibilidad de vectores) no ocurre así con los ordenadores personales actuales. Veamos pues la variante que utiliza arrays (en inglés array quiere decir literalmente formación; se usa con el significado de "vector", tabla y similares).

```
 5 INPUT MAX:DIM PR(MAX):PR(1)=1:
   PR(2)=2:K=3
10 FOR NP=3 TO MAX STEP 2
20 LIM=SQR(NP)+1
30 I=3
40 IF PR(I)>LIM THEN 70
50 IF NP=INT(NP/PR(I))*PR(I) THEN 80
60 I=I+1:GOTO 40
70 PR(K)=NP:K=K+1:CP=CP+1
80 NEXT NP
100 PRINT:PRINT "ENTRE 1 Y ";MAX;" HAY";
110 PRINT CP;" NUMEROS PRIMOS"
```

En vista de que este nuevo programa refleja muy de cerca el anterior, nos parece suficiente dejar la comparación como ejercicio útil y limitarnos a hacer notar que, poniendo los habituales 1 y 2 en los dos primeros puestos del array PR de los números primos, CP partirá desde cero, aunque también ahora se actúa con los NP impares y se hace el test desde PR(3) en adelante. Pero no hemos terminado. Los que disfrutan pensando, seguramente habrán oído hablar de la Criba de Eratóstenes.

En BASIC se podría expresar así:


```
 5 VERDADERO=1:FALSO=0:INPUT MAX:DIM PR(MAX)
10 FOR I=1 TO MAX:PR(I)=VERDADERO:NEXT I
20 FOR X=2 TO MAX
30 IF NOT PR(X) THEN 70
40 FOR Y=X TO MAX STEP X
50 PR(Y+X)=FALSO
60 NEXT Y
70 NEXT X
80 FOR I=1 TO MAX:IF PR(X) THEN PRINT PR(X):CP=CP+1:NEXT I
90 PRINT "HAY ";CP;" PRIMOS HASTA ";MAX
```

El procedimiento consiste en establecer inicialmente que todos los datos de una matriz de valores booleanos de amplitud MAX, son "VERDADERO" después de lo cual, partiendo de 2, se van a situar como "FALSO" todos aquellos cuyo índice sea múltiplo de cada número primo. La clave está en la línea 50: PR(Y+X) que proporciona (para X=2) los índices 4, 6, 8 (en el caso de X = 3 se tratará, en cambio de 6, 9, 12, etc.) Al acabar, la matriz-criba estará reducida a un colador, con valores "VERDADERO" (1) en los elementos primos, o sea:

VALORES:	1, 1, 1, 0, 1, 0, 1, 0, 0,	1, 0, 0, 0, 0...
INDICES:	1 2 3 4 5 6 7 8 9...	23 24 25 26 27...

¡Nótese que, esta vez, los números (primos o no primos) son los propios índices! Así, la impresión y cuenta de los primos (línea 80) consiste en considerar sólo los índices de los elementos "VERDADERO". Por último, insistir en la elegancia derivada de la utilización de los booleanos: IF NOT PR o IF PR son expresiones elegantes que IF PR = 0 ó IF PR = 1 (en los dialectos BASIC en los que el booleano "true" no es 1, bastará con establecer VERDADERO =-1).

El problema de los números primos es una confirmación de nuestra Primera Reflexión, pero si lo pensamos bien nos lleva también a la

SEGUNDA REFLEXION: cualquier algoritmo está íntimamente ligado a una estructura de datos (adecuada).

Esta idea, verdaderamente fundamental, es la base, entre otras, del texto de Niklaus Wirth (padre del lenguaje Pascal), con el significativo título siguiente:

"Algorithm + data structures = programs"

En efecto, es evidente que en los dos últimos ejemplos, sin la estructura "matriz" no habríamos podido hacer nada, o hubiéramos tenido que arreglarnos a duras penas con más trabajo y memoria.

Veamos ahora otro pequeño ejemplo sobre estos conceptos. Aún con su banalidad tiene un valor histórico: muestra como el software de aplicación evoluciona tanto por perfeccionamientos sucesivos como por el empuje de sucesos del mundo real. En el ejemplo, el cambio de las leyes fiscales. La tabla de la figura 4, representa una posible tasa progresiva del IRPF (Impuesto sobre la Renta de las Personas Físicas) representada por meses, para el cálculo de las retenciones. Supongamos ahora que nuestro ordenador no permita tablas con índice. A primera vista parecería inevitable recurrir a horribles cadenas de IF/THEN como las siguientes:

```
200 IF IMP<=25000 THEN IRPF=0.1*IMP
210 IF IMP>25000 AND IMP<=33333 THEN
    IRPF=2500+0.13*(IMP-25000)
220 IF IMP<33333 AND IMP>=41666 THEN
    IRPF=3583+0.16*(IMP-33333)
.......ETC.......
```

La poca elegancia de esta solución es indiscutible. Dicho sea de paso, hay que constatar que, incluso problemas tan banales, NO siempre encuentran solución en una fórmula, aun complicada. Entonces se requiere un algoritmo, un procedimiento por pasos, seguramente con algunas iteraciones.

TRAMO IMPOSITIVO DE RENTA IMPONIBLE	IMPUESTO MINIMO (BRUTO)	PORCENTAJE %
0	0	10
25.000	2.500	13
33.333	3.583	16
41.666	4.916	19
50.000	6.500	22
62.500	9.250	25
....	....	....

Figura 4.—Tabla de bases imponibles de una ficticia clasificación del IRPF, calculada por meses. Cuando la base imponible supera un tramo impositivo y entra en el siguiente se aplica el impuesto mínimo (contenido en la segunda columna) más el tanto por ciento progresivo sobre la diferencia.

¿No sería posible, incluso con las limitaciones señaladas, algo menos rudo? Si nos fijamos, observaremos que el aumento de los tramos impositivos de rentas sujetas a impuesto y el de los porcentajes presenta una cierta regularidad. En efecto, hasta 50.000 pesetas el incremento en el tramo es de 8.333 ptas, y luego de 12.500 ptas. En cuanto a los porcentajes, el aumento es todavía más regular: de 3 en 3 puntos. El truco consiste en aplicar el 10% del IRPF inicial (es el mínimo al que nadie escapa) al entero IMPonible y añadir un posterior 3% a la diferencia de esta cifra (IMP) con respecto a un término móvil PARAG. Este último, inicialmente igual a 25000, se aumenta cada vez en 8.333, hasta 49.999 (también el Fisco acepta un error de una peseta...) y, desde entonces, en 12.500. Todo esto quedaría en Pascal:

```
PROCEDURE irpf(impon:REAL;VAR tasa:REAL);
VAR parag,incr1,incr2:REAL;
BEGIN
 tasa:=0.1*impon;
 parag:=25000;incr1:=8333;incr2:=12500;
 WHILE impon>=parag do
  BEGIN
   tasa:=tasa+0.3*(impon-parag);
   IF parag<49999 THEN parag:=parag+incr1
    ELSE parag:=parag+incr2;
  END;
END;
```

La traducción BASIC es banal:

```
100 REM SUBRUTINA IRPF
110 TASA=0.1*IMP:PAR=25000
120 I1=8333:I2=12500
130 IF IMP<=PAR THEN 180
140 TASA=TASA+0.3*(IMP-PAR)
150 V=(PAR<49999):W=NOT V
160 PAR=PAR+V*I1+W*I2
170 GOTO 120
180 RETURN
```

Percátese de la "finura" booleana contenida en las líneas 150 y 160. Es apropiada para los dialectos BASIC en los que el valor lógico "true" está representado con "1" (en los que se tiene "-1" será necesario sustituir, en la línea 160 los "+" por "-". Las variables W y V son mutuamente exclusivas (cuando una es 0 la otra

es 1, y viceversa) así que cuando sea verdadera la condición (PAR <49.999) V=1 y W= 0 por lo cual, de hecho, sólo se añade el incremento I1 = 8.333 a "PAR". A continuación, los papeles de W y V se intercambian y lo que se añade a PAR es I2 = 12.500. Es un modo de sustituir la construcción IF/THEN/ELSE, de la que carecen muchos dialectos de ordenadores personales.

Por las apariencias el programa anterior podría parecer bien planteado, pero no es así. Su fragilidad queda al descubierto con el continuo aumento de la presión fiscal, dado que para bases imponibles más altas, la regularidad de la tabla sería una bendición.

Es evidente que, al final, este asunto se debería resolver con el uso de tablas de tipo "correlacionadas". En Pascal esto significa que hay que utilizar el tipo "record", subdividido en tres campos: RENT, MIN y PORCENT.

En BASIC nos arreglamos con tres matrices de igual nombre y para correlacionarlas (es sencillo: con el mismo índice...) tendremos que arreglárnoslas nosotros:

```
100 REM SUB IRPF CON. TABLAS CORRELACIONADAS
110 DIM RENT(20);MIN (20);PORCENT(20)
120 FOR I=1 TO 20
130 READ RENT(I),MIN(I),PORCENT(I)
140 NEXT
150 FOR I=1 TO 20
160 IF IMP>RENT(I) THEN NEXT I
170 IF I=21 THEN PRINT "ERROR":STOP
180 TASA=MIN(I)+(IMP(I)-MIN(I))*PORCENT(I)
190 RETURN
200 DATA 25000,0,0.1,33333,2500,0.13
210 DATA 41666,3583,0.16,50000,4916,0.19
220 DATA .....ETC.....
```

En las líneas a partir de la 200 se cargan los DATA correspondientes a las variables correlacionadas RENT, MIN y PORCENT. La primera terna, como se aprecia en seguida, está compuesta por el tramo impositivo de renta RENT = 25.000, por el impuesto mínimo MIN = 0 y por el tanto por uno (porcentaje ya dividido por 100) PORCENT = 0.1. Ni estas ni las otras ternas de los DATA coinciden con las líneas de la tabla incluida en la figura 3, contrariamente a lo que podríamos haber esperado. ¿Por qué? Es sencillo; para entenderlo basta con seguir el mecanismo de la búsqueda en tablas. En efecto, el proceso (que, naturalmente necesita una matriz-llave ordenada), funciona no "por igual" sino "por menor" y, en este caso, cuando se encuentra con una situación de IMP menor (o igual) a RENT tiene que aplicar los MIN y PORCENT que, en la figura 4, están en la línea precedente. Por lo tanto, merece la pena cambiar estas líneas tal y como se hizo en los DATA. Así, cuando nos encontremos IMP ≤2500 el IMP será 0 y el PORCENT = 0.1; cuando tengamos IMP ≤ 33333 MIN será = 2500 y PORCENT = 0.13, etc.

Una última observación. En teoría podríamos haber prescindido del vector MIN, dejando al ordenador la aburrida tarea de calcular las distintas bases mínimas, pero es más fácil proceder como lo hemos hecho. En cualquier caso, si decidimos que lo haga, deberemos evitar que el cómputo, por ejemplo, de las 3583 ptas (como suma del MIN anterior, igual a 2500, más el 13% de la diferencia entre PORCENTajes contiguos, MIN = 0.13*(33333-25000) + 2500 = 3583) cada vez que debamos utilizarlo, y hacerle realizar estos cálculos al principio y de una vez por todas.

El ordenador NO es una máquina pensada para realizar pequeños cálculos que ya podamos saber de antemano (podemos obtener los resultados con una calculadora en caso de que seamos tan vagos). Usarlo para esto es no sacarle partido.

Si, por ejemplo, necesitamos en muchos lugares del programa la raíz de 2, será conveniente calcular al principio una variable R2 = SQR(2) o bien meter, donde sea necesaria, la constante 1.4142135.

Abandonemos aquí los problemas de cálculo. En los próximos capítulos abordaremos temas que, esperamos, les ayudarán a comprender, en concreto, como hay que plantear y escribir un programa correctamente.

CAPITULO II

LABERINTOS, CLASIFICACIONES Y ESTADOS VARIABLES

Ariadna, su hilo y Teseo

De los tiempos del colegio nos llegan los recuerdos de los mitos griegos, todos ellos cargados de resonancias metafísicas y a veces angustiosas. Tampoco faltan los misterios, basta con pensar en Edipo delante de la Esfinge tratando de resolver su enigma, en tiempos en los que todavía no existían las revistas de crucigramas.

En cuanto a Teseo, sabemos que debió en parte su éxito a su sex-appeal que le permitió conquistar a Ariadna. Y, ¿qué fue lo que le dio esta bella doncella? Evidentemente, un algoritmo. Haciendo notar que para su seguridad, Teseo, se tendría que proveer del carrete de hilo de Ariadna para señalar los pasillos por donde pasara, en términos un tanto esquemáticos este algoritmo se puede expresar como sigue:

```
CASE nudo OF
    "Minotauro"   ASESINA-MONSTRUO
    "Ciclo"       REBOBINA-HILO
    "Virgen"      DESENROLLA-HILO
    "Ariadna"     STOP
    "Otro"        REBOBINA-HILO

END
```

Todo esto está escrito en un Pascal de "andar por casa". Se enumeran las condiciones que se pueden dar en el nudo (o habi-

tación del laberinto de la que salen dos o más pasillos) y, junto a cada una de ellas, la acción a cumplir. El significado de estas condiciones es:

- "Minotauro" en la habitación está el monstruo
- "Ciclo" el hilo de Ariadna se encuentra en un pasillo;
- "Virgen" ningún pasillo ha sido recorrido (excepto aquel de donde se viene).
- "Ariadna" en el nuevo nudo está la doncella, o bien estamos otra vez en la entrada.
- "Otro" ninguna de las condiciones anteriores.

No es difícil demostrar (y, a posteriori, resultaría algo intuitivo) que el algoritmo funciona; incluso en el caso de que el monstruo sea inaccesible (bien porque esté encerrado en una habitación, bien porque haya sido asesinado por otros) e implica además, la vuelta del héroe. Para nuestros lectores y utilizando términos informáticos, debemos mencionar que esta recuperación del hilo, recorriendo sus propios pasos, se llama "backtraking"; se desarrolla mediante una estructura de datos denominada pila (stack) en la cual se acumulan los datos para posteriormente extraerlos uno a uno, comenzado por el último introducido.

Esto podría sugerir a los más adelantados la realización de un pequeño programa que simule el juego del laberinto. Sin embargo, nosotros aprovechamos la ocasión para expresar la

TERCERA REFLEXION: cualquier programa refleja una situación de estados variables.

También se podría decir que el programa es una máquina de estados variables. Aquí resulta obligado citar la célebre Máquina de Turing, ordenador idealizado (como la Máquina de Carnot en Termodinámica). A cada paso sucesivo de elaboración el ordenador asume una determinada configuración o "estado"; las variables que están en la memoria toman entonces, sucesivamente, valores que, a su vez (por ejemplo, con la presencia de instrucciones condicionadas del tipo IF, CASE, ON... GOSUB, etc.), determinan el estado futuro. En todo este proceso el software es un poco el motor inalterable (por lo menos mientras se excluya el caso, novísimo de los programas que se automodifican) y la secuencia de instrucciones determina, a priori, el movimiento total. En la práctica, esto significa que al programar hay que desarrollar dos tareas:

- comprender cómo se tiene que mover la máquina de estados variables;
- prever en el programa las reglas adecuadas.

O sea, programar significa prever, o mejor, escribir, lo que debe hacerse bajo todos los puntos de vista, sin olvidar que el ordenador es una máquina secuencial, es decir, que ejecuta las instrucciones de una en una. La historia de Teseo y Ariadna es significativa respecto a esto.

Para concretar de forma sencilla el concepto, examinemos ahora un laberinto simplificado como el de la figura 1. El objetivo es el siguiente: se trata de realizar un juego del tipo "aventuras", en el que el usuario tiene que llegar a la habitación del tesoro (T) y volver a la entrada. Esta vez, por lo tanto, la tarea de no perderse (¡y sin el hilo de Ariadna!) es responsabilidad del hombre, mientras que el ordenador se limita a hacer de notario. La máquina de estados variables que hemos utilizado puede representarse con el grafo de la figura 2.

¿Qué es un grafo? se preguntarán algunos de ustedes. Antes de contestar queremos precisar que cada habitación puede tener hasta 4 puertas, tantas cuantos puntos cardinales, y que algunos accesos pueden estar cerrados o tapiados. Dicho lo cual podemos aclarar que un grafo representa, con un círculo pequeño (nudo) cada estado del sistema; los "arcos" orientados, cada uno señala-

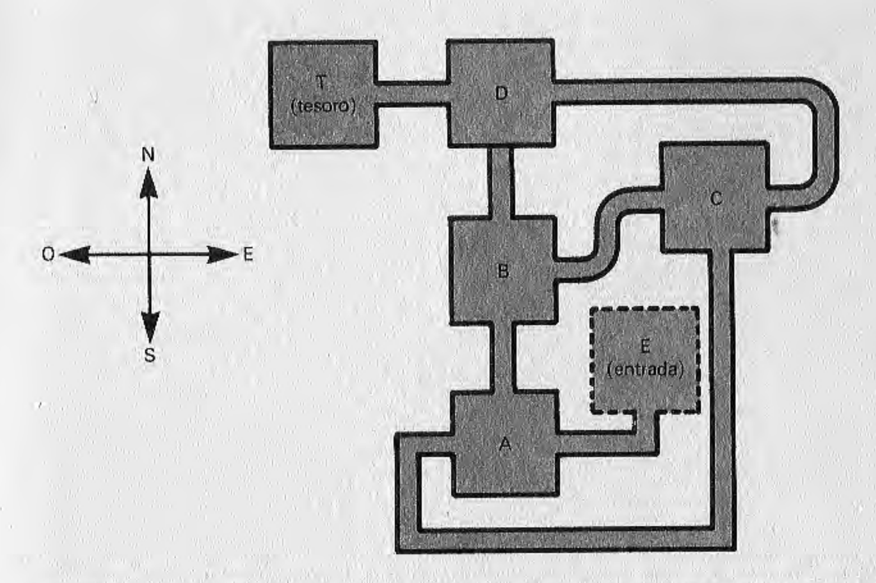


Figura 1.—El sencillo juego del laberinto. Para mayor sencillez las puertas sólo tienen 4 orientaciones; admite la posibilidad de volver a entrar por una dirección distinta a la usada cuando salimos, a causa de la tortuosidad de los pasillos.

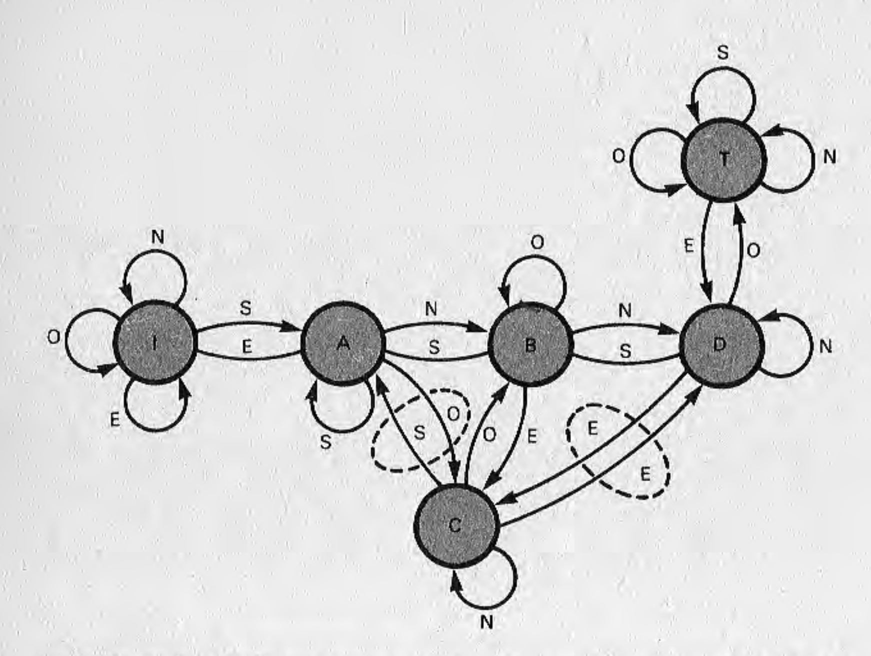


Figura 2.—Grafo que representa la máquina de estados variables del laberinto. Cada nudo es una habitación, cada arco (transición) corresponde a un pasillo. Las parejas encerradas por trazos indican la posibilidad de volver a entrar en un sentido cualquiera.

do con la clave de la acción asignada, expresan las posibles "transiciones" de un estado a otro. Así, en la figura citada, cada nudo representa una habitación, mientras que las cuatro flechas que salen de allí representan los pasillos y corresponden a las respectivas elecciones hechas por el usuario (terminan en la habitación a la que da paso el respectivo pasillo). Cuando el pasaje está cerrado, la flecha vuelve al mismo nudo, evidenciando que el estado no ha cambiado. Nótese también, comparando las figuras 1 y 2, que la flecha de vuelta no siempre tiene el sentido opuesto al de ida, desde el punto de vista de la rosa de los vientos; por ejemplo, de A se va a C desde el oeste pero desde C se vuelve a A por el sur. Esto depende de la tortuosidad de algunos pasillos, y debería de aumentar la desorientación según los planes del diabólico arquitecto Dédalo.

En este ejemplo la correspondencia entre nudos-estados y arcos-pasillos es estrechísima, pero en otros puede ser más abstracta; por ejemplo, pueden hacer corresponder a un nudo el estado de una centralita automática de telecomunicaciones y, a un

arco, la llegada de una señal particular, conforme a un cierto "protocolo" de comunicación que hará reaccionar a la primera de una forma determinada. Ciñéndonos al tema de los juegos, en los de tipo estratégico cada nudo hace evidente estados en los que se dan cosas (como la cantidad de dinero, la felicidad, las naves espaciales) poseídas por cada jugador, el número de bienes o planetas conquistados, etc., mientras las elecciones (flechas orientadas), están ligadas a las acciones decididas (como el envío de naves a otros planetas, la compra de acciones de Telefónica o similares, etc.). Se puede complicar el juego a voluntad, pero el concepto es, en esencia, idéntico.

Ordenación por "burbuja"

Vamos ahora a dejar un tiempo de reflexión para que los lectores piensen sobre lo anteriormente expuesto. Solo comentaremos antes, para los más inquietos e impacientes, que un grafo del tipo que hemos visto es equivalente a un corrientísimo diagrama de flujo y que, en general, cada flecha orientada puede ser sustituida por un IF THEN... (por lo menos en teoría: anticipamos que hay una solución más elegante, pero no diremos más).

En esta especie de intervalo nos ejercitaremos en el tema de la Primera Reflexión, relacionada con el carácter evolutivo del software. Tomemos pues el diagrama de flujo de la figura 3, referente a la clásica ordenación por el "método de la burbuja" (en inglés "bubble sort") aplicada a una matriz de números A. El método se denomina así porque las translaciones que sufren los datos recuerdan el movimiento ascendente de las burbujas en líquido. El programa correspondiente en BASIC sería como sigue:

```
5 INPUT N:REM El usuario especifica el numero de elementos
10 DIM A(N)
20 FOR I=1 TO N
30 A(I)=INT(RND(1)*1000+1):PRINT A(I);" ";
40 NEXT I
50 FOR I=1 TO N-1
60 IF A(I)<=A(I+1) THEN 80
70 C= A(I):A(I)=A(I+1):A(I+1)=C:W=1
80 NEXT I
90 IF W THEN W=0:GOTO 50
100 PRINT:PRINT
110 FOR I=1 TO N:PRINT A(I);" ";:NEXT I
120 END
```

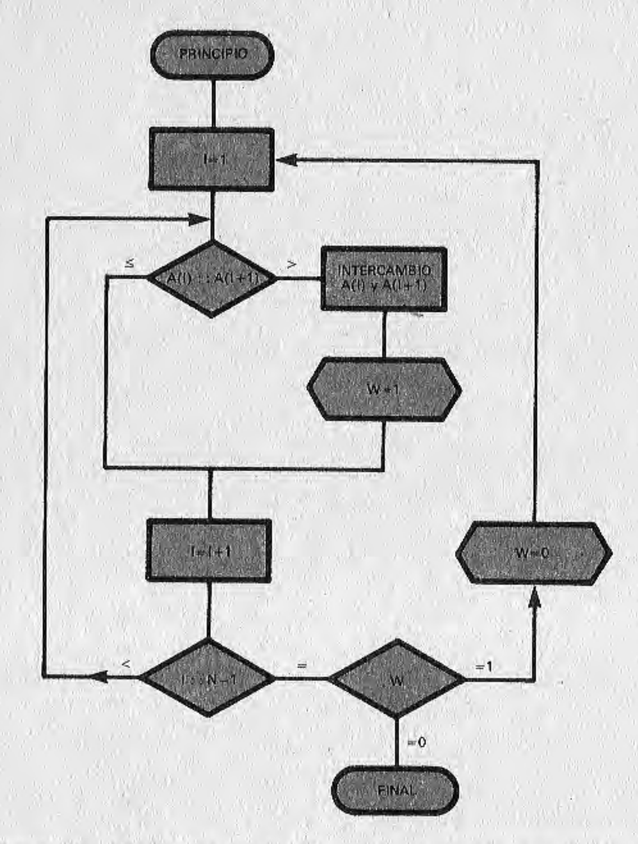


Figura 3.—Diagrama de la clásica "ordenación por burbujas" (Bubble Sort).

Las líneas 30 y 110 imprimen los números de la lista (generados al azar en el campo de los enteros desde 1 hasta 1000) antes y después de la operación. El ordenamiento (o "reordenamiento") con el algoritmo de las burbujas consiste, como quizá alguien ya sepa, en examinar de dos en dos los elementos contiguos, de índice I e I + 1. Si están descolocados (aquí la hipótesis es que se desea un orden ascendente) se les cambia usando una variable auxiliar C y además se activa una variable booleana W (tenga presente que en algunos de los dialectos de BASIC el valor lógico "verdadero" corresponde a −1, en vez de a 1). Todo esto está en la línea 70. Al acabar el ciclo FOR/NEXT se chequea W y si está en "on" se vuelve a empezar el ciclo para dar otra pasada de posibles cambios. Si, en cambio, W es 0 quiere decir que cada ele-

mento estaba seguido por otro de un valor mayor, por lo que todo está correcto y el ordenamiento cesa.

Para estudiar otras variantes y, razonando, poder llegar a soluciones alternativas, posiblemente mejores, proponemos aquí dos ejercicios. Para que sirva de "calentamiento" empezaremos con una pregunta muy sencilla: ¿es el desviador W verdaderamente indispensable? Respuesta: en absoluto. En la línea 70 es suficiente sustituir W = 1 por un simple GOTO 50 (y, naturalmente, eliminar la línea 90, que ahora resulta inútil). Esto equivale, sin embargo, a tener que empezar desde el principio cada vez que se hace un intercambio, de forma que, cuando se sale del NEXT, todo esté en su sitio.

He aquí los dos ejercicios mencionados:

1. realice el programa usando dos índices distintos (I y J), haciendo rotar más rápidamente el segundo que el primero (por lo tanto, dos ciclos FOR/NEXT anidados uno en el otro) y comparando sistemáticamente A(I) con A(J) para intercambiarlos si el primero es mayor;
2. comparando siempre elementos adyacentes, pero, cuando se encuentre una pareja fuera de sitio, realice una serie de intercambios hacia atrás, hasta llegar a una pareja bien colocada, volviendo entonces a barrer hacia delante "desde el punto en que nos habíamos quedado".

Lo importante es verificar si, con estas variantes, se consigue ganar algo de velocidad. En efecto, el ordenamiento de burbuja se vuelve muy lento al aumentar el número de elementos. Para ser más exactos, se demuestra que la media de tiempo de un ordenamiento de este tipo crece con el cuadrado de la dimensión (o "potencia") del conjunto, por lo que doblándolo se producen tiempos cuádruples.

La primera sugerencia que les damos es la de intentar aplicar, de alguna forma, el principio del divide y vencerás. La receta es sencilla:

1) se hace la clasificación de dos mitades en matrices distintas;
2) se realiza la fusión de las dos semimatrices reordenadas.

Este proceso está ilustrado en la figura 4. En *a*)se representa la subdivisión elegida (entre elementos de lugar par e impar). En *b*) se representa la situacion después de los reordenamientos parciales (de burbuja) sobre los números originales (como en a). Nótese también que, siendo en este caso la potencia igual a 9, se ha añadido un décimo elemento de valor HV. Sirve para cuadrar las

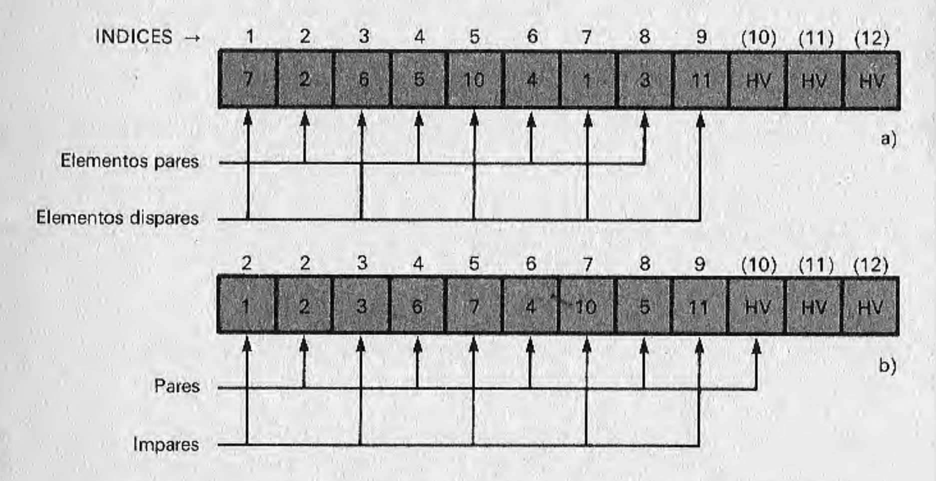


Figura 4.—Situación de la matriz que debe reordenarse, antes a) y después b) de la semi-ordenación de los elementos con puestos pares e impares. Nótese el "tapón" HV añadido en el décimo lugar para cuadrar el número de elementos.

cuentas, haciendo así que la semimatriz par tenga cinco elementos, al igual que la impar. Pero, ¿qué significa HV? Viene del inglés "High Value" (valor alto); se utiliza como "tapón" al final de un conjunto (o file) y tiene que asumir un valor mayor que el más alto posible del conjunto. En nuestro pequeño ejemplo se utilizan valores de 1 a 1000, por lo que bastará poner HV = 9999, mientras que en el caso de datos alfabéticos será suficiente con HV$="ZZ" (mejor aún, el mayor número de "Z" posibles) para estar tranquilos. En el caso de una clasificación decreciente se hablará de los correspondientes LV (Low Values) por ejemplo -999...9). Pero dejemos hablar al listado:

```
5 HV=9999
10 INPUT N:DIM A(N+3),B(N+1)
20 FOR I=1 TO N:A(I)=INT(RND(1)*1000+1):NEXT I
30 IF INT(N/2)*2<N THEN N=N+1:A(N)=HV
40 A(N+1)=HV:A(N+2)=HV
50 GOTO 120:REM Salta a la rutina 60,
   de clasificacion de burbuja
60 FOR I=X TO Y-1 STEP 2
70 IF A(I)<=A(I-1) THEN 90
80 C=A(I):A(I)=A(I-1):W=1
```



```
90 NEXT I
100 IF W THEN W=0:GOTO 60
110 RETURN
120 REM Reordenacion de los elementos pares
130 X=2:Y=N:GOSUB 60
140 REM Reordenacion de los impares
150 X=1:Y=N-1:GOSUB 60
160 ...(Sigue mas adelante)...
```

En la línea 30 se hace un test de paridad: si, por ejemplo, N = 9 la INT (N/2)*2 nos da 8<9 por lo que se añadirá un elemento de valor HV=9999. En cuanto a la clasificación por burbuja, está descrita bajo la forma de una subrutina "paramétrica" (desde la línea 60 a la 110). Los parámetros son las variables "X" e "Y" (valores iniciales y finales de la submatriz); aparte del STEP 2 (que es obvio, ya que se tiene que trabajar con los pares o con los impares) todo continúa como hemos visto. Es en las líneas 130 y 150 donde los ya citados parámetros se fijan para que la subrutina de la línea 60 haga su trabajo de reordenación de los elementos pares e impares; haciendo esto, hemos logrado escribir las instrucciones adecuadas una sola vez. Una vez llegados a este punto debemos realizar la fusión de las dos semi-matrices en una única matriz B. Hela aquí:

```
160 REM Fusion (MERGE)
170 X=1:Y=2
180 FOR Z=1 TO N
190 IF A(X)>A(Y) THEN B(Z)=A(X):X=X+2:GOTO 210
200 B(Z)=A(Y):Y=Y+2
210 NEXT Z
220 FOR I=1 TO N:PRINT B(I);" ";:NEXT I
230 END
```

La fusión resulta banal. Presupone dos archivos de partida (barridos con distintos índices "X" e "Y") ordenados ambos del mismo modo; procede siempre con ordenación ascendente, usando la matriz que tiene el elemento menor e incrementando el índice de éste. En cuanto a los "tapones" (otros los llaman "centinelas"), sirven para mantener el sentido de la comparación entre elementos A(x) y A(y), hasta el final. De esta manera, al alcanzar el elemento valor 9999 en una semi-matríz, el programa completará automáticamente la carga de los elementos residuales de la otra, todos inferiores a 9999. Si no hubiésemos adoptado el criterio de los tapones, la casuística se habría complicado bastante; basta exami-

nar la figura 5 para creerlo; en *a*) y en *b*) se han representado los flujos de la fusión con y sin "tapones". La figura demuestra que —aunque hay gustos para todo— quienes prefieran evitar soluciones demasiado complicadas, también podrían seguir adelante.

Hablando de gustos, es muy probable que aquí alguien pregunte: ¿qué necesidad había de subdividir los elementos en pares e impares? ¿no era más sencillo coger simplemente la primera y la segunda mitad? Efectivamente, pero haciendo esto el truco de los tapones no se hubiera podido pliacar, a menos que nos deslicemos un lugar a la izquierda a la mitad de la formación: ¡menudo adelanto para quien persiga fines de velocidad!

Después de experimentar la reducción de tiempos conseguida con este método (a pesar del añadido de la fusión, con una docena de elementos ya se nota que es un proceso cuya ejecución depende linealmente del número de elementos en juego) recordaremos fugazmente que la clasificación, junto a la fusión y al "search" (búsqueda de un elemento determinado), constituye uno de los capítulos más densos de la informática. Subrayemos dos cuestiones que de aquí se deducen:

- el carácter evolutivo del software y de los algoritmos;
- la importancia de una estructura de datos adecuada.

Es decir, es una repetición de las que hemos llamado Primera y Segunda Reflexión.

En relación con la primera animamos a los más adelantados a afianzarse en la creación de otros algoritmos derivables, por así decirlo, de los primeros.

Otros dos tipos de ordenación

Para completar el tema mínimamente es obligada la alusión a otros dos tipos de ordenación "clásicos": la clasificación por mínimos y la clasificación rápida. El primero se parece a la "burbuja" pero es un poco más rápido; consiste en barrer al primer giro todo el conjunto, encontrando el mínimo, que será puesto en el primer lugar. Después se repite el procedimiento con los N−1, N−2...2, elementos sucesivamente residuales.

En cuanto a la clasificación rápida, debe su nombre al estupor de su inventor Hoare, ante su velocidad. El procedimiento "P" consta de los siguientes pasos:

- encontrar el elemento del centro de la matriz: sea X su valor;

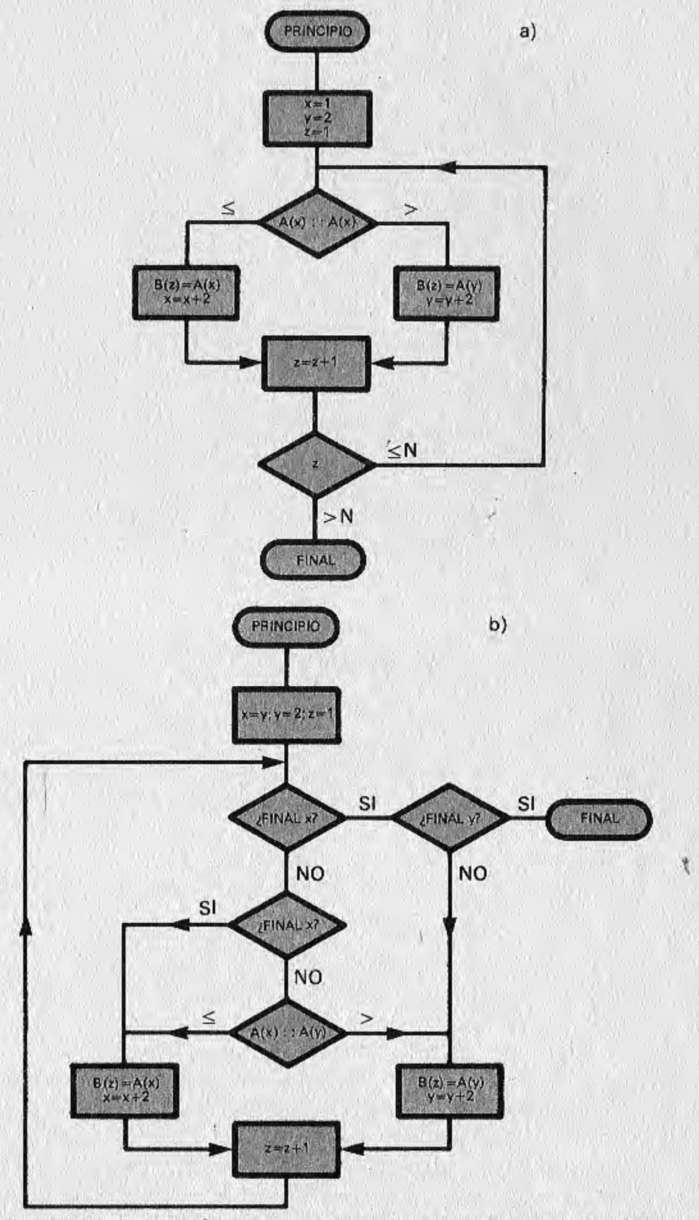


Figura 5.—Fusión de dos matrices. La primera solución (a) se realiza mediante la adopción de los tapones HV, y es más sencilla que la otra (b) en la que se tienen que tener en cuenta las distintas posibilidades de un final parcial para cada matriz.

- buscar desde el primer elemento en adelante un elemento mayor que X;
- buscar desde el último elemento hacia atrás, un elemento menor que X;
- cambiar de sitio los dos elementos encontrados;
- continuar hasta que (de derecha o izquierda) se llegue al elemento central.

La verdadera clasificación rápida consiste en la aplicación reiterada de P en la matriz entera, después en sus dos mitades, en las varias mitades de la mitad y así sucesivamente, hasta llegar a porciones de tan solo dos elementos. ¿Complicado? Bueno, es por ello que aconsejamos algún texto especializado para profundizar en ella, mientras que, para ejercitarnos, nos limitaremos a la clasificación de los mínimos, que es más fácil. De todas formas, los que sepan profundizar en la materia, se darán cuenta que, siendo testarudos, con las matrices se puede llegar lejos pero hasta un cierto punto; sólo pensando en nuevas estructuras se puede intentar optimizar otras funciones, entre las cuales está la puesta a día o la cancelación de un elemento, o la suma de uno nuevo en el lugar adecuado. Ponemos aquí un ejemplo (figura 6) aludiendo a la estructura de árbol binario. En ella cada "nudo" contiene un dato y es "padre" de un nudo izquierdo menor y de un nudo derecho mayor. El parentesco está fijado por los punteros correspondientes, que son campos de cada nudo que indican la colocacón de los hijos derecho e izquierdo. Esta estructura hace particularmente fácil la puesta al día. Por ejemplo, al añadir un dato, no hace falta, como con las matrices, reordenar toda la formación, perdiendo así un montón de tiempo; basta "visitar" el árbol para saber dónde hay que colgar el nuevo hijo; en la figura 6b) se muestra, con línea de trazos, la visita que hay que hacer para colocar un nuevo dato (13) que hay que añadir al árbol de a).

Volviendo al laberinto

Pero bajemos de los árboles y de los frutos que se podían coger de sus frondosas e intrincadas ramas. Habrán notado que, en los ejemplos del "entreacto", no hemos traído a colación la Tercera Reflexión, la de la máquina de estados variables. En realidad sí apareció, pero sobrentendida, intrínseca a la naturaleza misma de cada algoritmo. En cambio, en problemas como el del laberinto, la idea de la máquina de estados variables, más que útil es indispensable. Aquí nos proponemos mezclar adecuadamente la Tercera Reflexión con la Segunda.

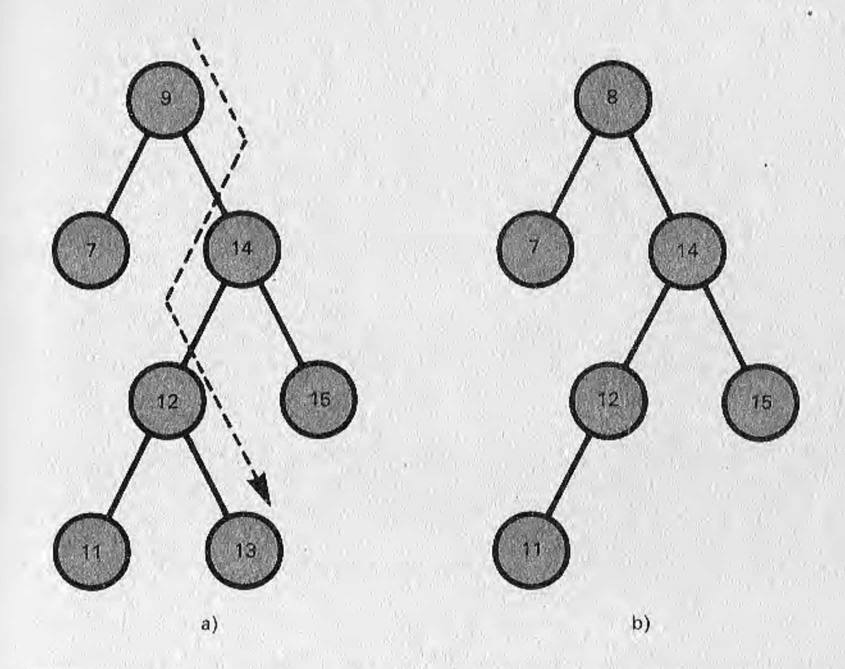


Figura 6.—Estructura de datos general en un árbol binario. Los elementos mayores y menores son los "hijos" derecho e izquierdo de cada nudo (utilizando punteros). En b) se observa "la visita" que hay que realizar para encontrar la colocación de un nuevo dato (13).

Dado que ya hemos dibujado el grafo con los estados y sus posibles transiciones, la pregunta que debemos hacernos es: ¿qué estructura de datos puede ser más conveniente para describir nuestra máquina de estados variables y su desarrollo? Los apresurados y los principiantes probablemente lo resolverían así (expresándolo en pseudo-Pascal):

```
..........
IF habitacion-A AND resp:='N' THEN
   habitacion-B
ELSE IF habitacion-A AND resp='S' THEN
    WRITE ('no hay camino')
.
.
ELSE IF habitacion-B AND resp='N' THEN
   habitacion-D
..........
```

Los más refinados quizá adoptarían una construcción CASE, más elegante, pero sería igualmente una chapuza, aunque funcionase, al menos por dos buenos motivos:

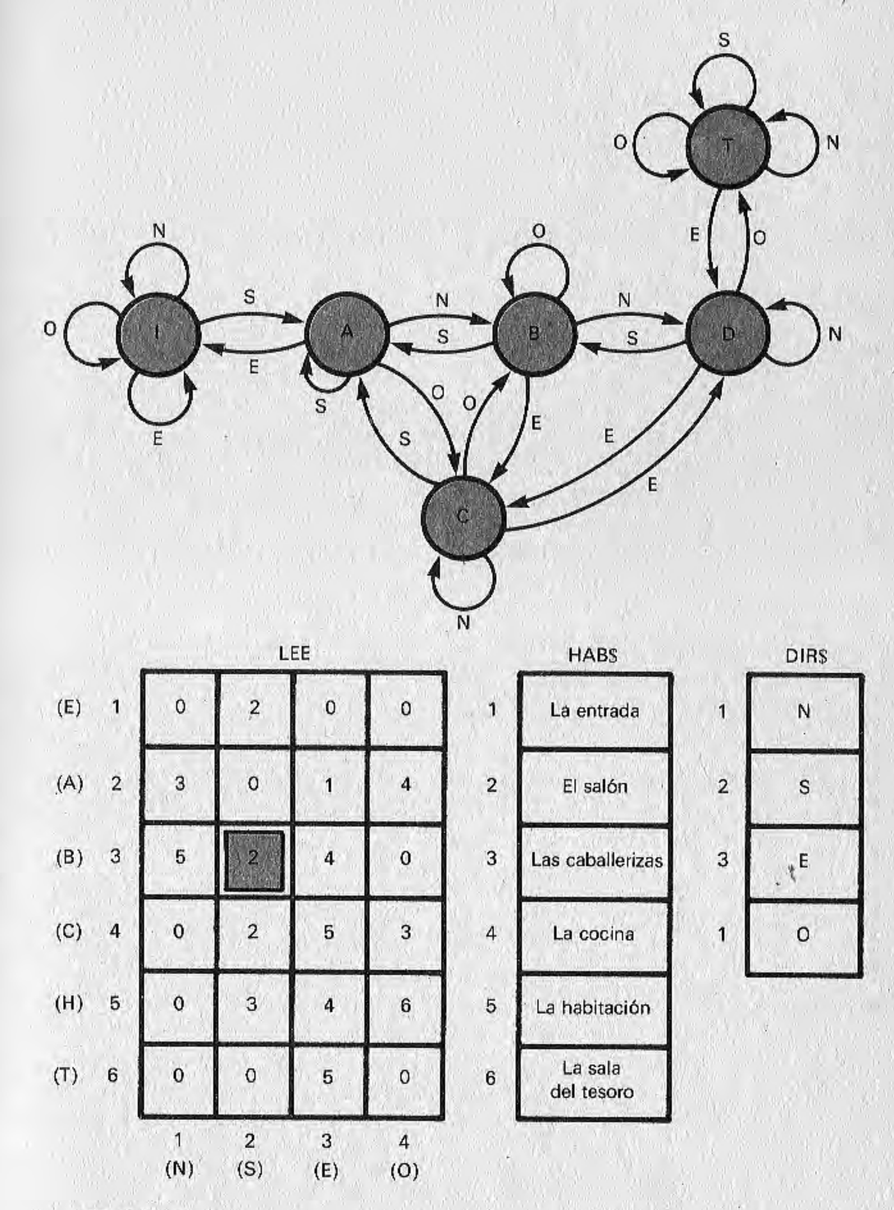
- en cuanto el laberinto se vuelve un poco complicado la longitud y legibilidad del conjunto se van al garete;
- algo todavía más trágico: si el demonio nos hace añadir una sola habitación más, nos arriesgamos a tener que escribir todo desde el principio otra vez.

En definitiva: es necesaria una verdadera estructura de datos. La que proponemos está ilustrada en la figura 7, donde, por comodidad, hemos vuelto a dibujar nuestro grafo. El meollo de la cuestión es la tabla de dos dimensiones (o "matriz", como normalmente se la llama) denominada UNI (por UNIones), cuyas filas representan las diferentes habitaciones, mientras que las columnas están asociadas a los 4 puntos cardinales. Por ejemplo, en la intersección de la línea 3 con la columna 2 se encuentra el valor UNI (3,2) = 2, que indica que si nos encontramos en la habitación 3 (la B del grafo) y elegimos la puerta situada al Sur llegaremos a la habitación 2 (la A del grafo). Al paciente lector le será fácil encontrar todas las otras correspondencias, evidentes en las diversas casillas. En esencia estas contienen "punteros"; obviamente, un puntero 0 corresponde a las flechas del grafo cerradas sobre sí mismas (puertas tapiadas). En cuanto a los vectores alfanuméricos ST$ y DIR$ sirven, el primero para asociar las direcciones 1, 2, 3 y 4 en el orden: N, S, E y O, (aunque también se podrían prever los nombres enteros) y, el segundo, para asignar un nombre de fantasía —"ENTRADA", "CUADRA", "SALA TESORO", etc.— a los diferentes ambientes del minilaberinto.

No nos queda ahora más que volver a formular el problema: redactar un programa que, tomando las elecciones del usuario (N, S, E y O) le señale, cada vez, los trazos (pasillos) TI recorridos a la ida y los TV recorridos a la vuelta de la habitación del Tesoro (T), partiendo de la habitación de entrada (E).

Damos por descontado que lo intentarán ustedes. Todo resulta más sencillo una vez que se ha definido la máquina de estados variables y la estructura de datos. No autorizamos a los más perezosos a que miren la solución que estamos a punto de dar, sin intentarlo antes al menos.

```
 1 REM DATAS Y ASIGNACIONES INICIALES
10 DATA 0,2,0,0,3,0,1,4,5,2,4,0
20 DATA 0,2,5,3,0,3,4,6,0,0,5,0
30 DATA "LA ENTRADA","EL SALON","LA CUADRA"
40 DATA "LA COCINA","LA ANTESALA","LA SALA DE TESORO"
```

LEE

		1 (N)	2 (S)	3 (E)	4 (O)
(E)	1	0	2	0	0
(A)	2	3	0	1	4
(B)	3	5	2	4	0
(C)	4	0	2	5	3
(H)	5	0	3	4	6
(T)	6	0	0	5	0

HABS

1	La entrada
2	El salón
3	Las caballerizas
4	La cocina
5	La habitación
6	La sala del tesoro

DIR$

1	N
2	S
3	E
1	O

LEE (3,2) = 2

Figura 7.—La estructura de datos, adecuadamente organizada, refleja la máquina de estados variables del laberinto, haciendo además más limpio y general el programa.

```
50 DATA "N","S","E","O"
100 DIM UNI(6,4),ST$(6),DIR$(4)
110 FOR H=1 TO 6
120 FOR K=1 TO 4:READ UNI(H,K):NEXT K
130 NEXT H
140 FOR I=1 TO 6:READ ST$(I):NEXT I
150 FOR I=1 TO 4:READ DIR$(I):NEXT I
160 REM
200 REM INICIALIZIACIONES DIVERSAS
210 SW=0:H=1:TI=0:TV=0:HF=6
220 CLS:PRINT "ESTAS EN";ST$(H)
225 REM
230 REM EXAMEN DE CONDICIONES-LIMITE
240 IF SW=1 AND H=1 THEN GOSUB 3000:
    GOTO 260:REM FINAL
250 GOTO 290:REM CASO NORMAL
260 INPUT "¿QUIERES INTENTAR OTRA EXPLORACION? ";R$
270 IF LEFT$(R$,1)="S" OR LEFT$(R$,1)="s" THEN 200
280 END
290 IF H=HF THEN GOSUB 2000:
    REM ¡TESORO ALCANZADO!
295 REM ELECCION DEL USUARIO
300 REM
310 INPUT "¿A DONDE QUIERES DIRIGIRTE? (N,S,E,O) ";R$
320 FOR K=1 TO 4
320 IF LEFT$(R$,1)=DIR$(K) THEN 360:
    REM ENCONTRADA COLUMNA
330 NEXT K
350 GOTO 300:REM SINTAXIS ERRONEA,REPITE INPUT
360 IF SW=0 THEN TI=TI+1
370 IF SW=1 THEN TV=TV+1
380 IF UNI(H,K)<>0 THEN 410
390 PRINT "NO SE PUEDE IR HACIA EL ";R$
400 FOR R=1 TO 2000:NEXT R:GOTO 220:
    REM PAUSA Y AL PRINCIPIO
410 H=UNI(H,K):GOTO 220:REM RETOMA EL CICLO
420 REM INICIO SUBRUTINAS
1990 REM TESORO CONSEGUIDO
2000 SW=1:PRINT:PRINT "¡¡EL TESORO ES TUYO!!"
2010 REM OTRAS INSTRUCCIONES AD LIBITUM
1980 RETURN
```



```
2990 REM FINAL
3000 CLS:PRINT "FIN DEL JUEGO"
3010 PRINT:PRINT "HAS RECORRIDO ";TI;" PASILLOS ";
3020 PRINT "A LA IDA Y ";TV;" A LA VUELTA"
3030 REM OTRAS INSTRUCCIONES AD LIBITUM
4000 RETURN
```

Más corto de lo esperado ¿verdad? El mérito de que sea tan compacto viene dado seguramente por la extrema sencillez de la micro-aventura. Incluso hay que admitir que al programa le faltan algunas "cartas" para constituir verdaderamente el esqueleto de un juego de "aventuras" más desarrollado: sería necesario añadir cosas como un interface más cuidado con el usuario y, además, hacerlo más interesante en determinados aspectos (por ejemplo, situaciones de peligro, enigmas y similares) no todos fácilmente reconvertibles a estructuras de datos del tipo de los que aparecen en la figura anterior (aunque... ¿quién sabe?). Por otro lado, este problema se refiere, realmente, a los límites de la ingeniería del Software, sobre todo en su eterna búsqueda de la llamada "reutilidad" de los programas. La práctica demuestra, además, que casi en el 70% de los casos en los que hay que hacer frente a un problema nuevo, aunque sea parecido a los precedentes, es necesario volver a escribir de nuevo gran parte del programa.

Aún así, la solución propuesta permite añadir habitaciones y hacer variaciones, a voluntad, de las uniones recíprocas: es suficiente con modificar adecuadamente los DATA y las líneas iniciales, en las que se cargan los parámetros en las distintas matrices. Y, sobre todo, se han proporcionado algunas ideas que podrán ser reutilizadas, oportunamente cambiadas, en contextos parecidos, o incluso, a primera vista distintos. En suma, el saber no ocupa lugar, y esto también se aplica al moderno Arte de la Programación.

En cuanto a la explicación del programa nos limitaremos a lo esencial para no ofender a la mayoría de ustedes, que se supone conocen el BASIC (y si no, basta que acudan a los volúmenes 5, 6 y 7 de la BBI). Será suficiente seguir paso a paso las instrucciones para comprender lo que va sucediendo. La línea "clave" es la 410:

```
410 H = UNI(H,K): GOTO 220
```

El índice H, correspondiente a la línea de la matriz UNI, constituye, en la práctica, el estado de nuestra máquina de estados variables; con la instrucción ya vista, se extrae de UNI el nuevo valor (la nueva habitación), a la que nos lleva la dirección (columna) elegida por el usuario. En cuanto al índice K de esta columna,

determinado según la respuesta R$ dada en la línea 310 en términos alfanuméricos (N, S, E y O, o bien Norte, Sur, Este, Oeste, pues sólo cuenta la primera letra extraída con LEFT$(R$,1) en la línea 330), se establece numéricamente desde las líneas 320 hasta la 340. La búsqueda falla (no hemos contestado con un punto cardinal) si el ciclo FOR...NEXT es completado, mientras que tiene éxito si se interrumpe anticipadamente, lo que significa que el primer carácter de R$ es uno de los que están contenidos en el pequeño vector DIR$ (N, S, E y O).

Terminamos con estas pequeñas anotaciones: CLS, comando de limpieza de pantalla, puede sustituirse por el equivalente en otros BASIC (por ejemplo HOME, en el de Apple). El bucle de retardo, realizado con un bucle "ocioso" en la línea 400, sirve para mantener el tiempo necesario la frase "NO SE PUEDE IR HACIA EL"; R$. La variable HF, inicialmente igual a 6, determina, sobre la condición H = HF, la llegada al tesoro; en el caso de una ampliación del juego, admite la posibilidad de variar su valor, para ajustarlo al del nuevo juego, únicamente al principio (línea 210). Las REM de las líneas 2010 y 3030 sirven para recordar que las respectivas subrutinas pueden ser cambiadas a voluntad (por ejemplo, para añadir gráficos y/o sonoros). Al jugar con el programa alguno de ustedes notará que, si vuelve a entrar por segunda vez en la habitación del tesoro, se repite el mensaje "EL TESORO ES TUYO", que puede resultar trágico para quien quizá, se esté muriendo de hambre intentando encontrar la salida. A quienes no les guste esto, podrán remediarlo sin mucha dificultad.

Por último, la variable booleana SW, que se activa cuando se alcanza el tesoro (línea 2000), permite determinar cómodamente en la línea 240 el final del juego: éste termina cuando nos encontremos en la habiación 1 con SW = 1, condiciones que testimonian el final de la pesadilla.

CAPITULO III

LOS JUEGOS CON ESTRATEGIAS GANADORAS

La selva informática de los juegos

La informática revela un insospechado parentesco con la botánica, aunque sólo sea porque el objeto que encontramos con mayor frecuencia es el árbol, muchas veces puesto al revés con la copa hacia abajo y las raíces hacia arriba. Ya hemos hablado fugazmente de los datos. Ahora, después de haber sugerido qué estructuras están a la orden del día en los sistemas operativos (por ejemplo en el célebre sistema operativo Unix, la estructura de los archivos es arborescente) pasaremos a tratar otra importante utilización de los árboles: aquella en juego... en la Teoría de los Juegos y ¡nada más y nada menos! en la nueva y fascinante disciplina de la Inteligencia Artificial. Entre los juegos más sencillos que toman en consideración esta teoría, están aquellos en los que dos jugadores se alternan en los movimientos, siguiendo las reglas que definen su propia licitud y las situaciones de victoria, derrota y empate.

Ahora bien, se puede demostrar fácilmente que en muchos de estos juegos, a los que podemos llamar mecánicos, también podemos contradecir al sentido común, pues uno de los dos jugadores está predestinado a ganar o, por lo menos, a obtener una igualada (en aquellos que admitan un empate). Efectivamente, solo si se equivoca, ignorando la "estrategia vencedora" que está a su alcance, permite al contrario disponer a su vez de otra estrategia vencedora. Las damas, el ajedrez, el tres en raya y otros muchos pertenecen a esta categoría, y solamente la astronómica cantidad de combinaciones en juego en los dos primeros impiden co-

nocer para ellos una sola estrategia matemáticamente ganadora; ni siquiera se sabe si tal estrategia es privilegio del que empieza o del que juega en segundo lugar.

Un juego con estrategia vencedora: el Nim o Marienbad

Para entender mejor este capítulo, pondremos un ejemplo clásico que será objeto de un programa bastante instructivo e incluso, creemos, original.

El juego del Nim es un juego matemático suficientemente sencillo como para que se pueda delinear una estrategia ganadora. Hay quien lo llama "Marienbad" (en realidad Nim y Marienbad son el mismo juego, pero con reglas opuestas). Para simplificar no haremos ninguna distinción entre los dos. El juego en cuestión era un motivo repetido a menudo en la película del director francés Alain Resnais "L'annèe derniére á Marienbad" (El año pasado en Marienbad), una película misteriosa de la Nouvelle Vague. En ella el protagonista, G. Albertazzi, perdía sistemáticamente en este juego de cerillas —en los ricos salones de la estación termal— con el marido de su amante. ¿Desafortunado en el juego, afortunado en amores? ¡Qué va! La cuestión era simplemente, que el enigmático y poco fascinante adversario, conocía la clave de la estrategia ganadora del Nim, la misma que nos permitirá programar el ordenador de manera que la máquina venza inexorablemente (y sin la consolación de contrapartidas eróticas).

El Marienbad consiste en varias filas de objetos (palillos, cerillas, botones, etc.). En la figura 1 vemos, a la derecha, la configuración que se adopta habitualmente, mientras que a la izquierda está la que utilizamos en nuestro programa. Como veremos después, hay infinitas combinaciones posibles y el mismo número de juegos, ganados unos por quien empieza y otros por quien mueve en segundo lugar.

Las reglas son muy simples: los jugadores se alternan en las jugadas y, en cada una, pueden coger desde una hasta todas las cerillas de una misma fila. Pierde quien tiene que coger la última.

Examinemos un Marienbad muy simple (figura 2-a) y llamémoslo 3-2-1, por el número de piezas de sus filas. El árbol del juego está desentrañado en la misma figura, en b. Está claro que se trata de un árbol ramificado: la raíz constituye la configuración inicial y las ramas las sucesivas jugadas, cada una de las cuales conduce a la nueva configuración. El árbol es un grafo particular de la máquina de estados variables del juego. En realidad, por motivos de espacio y de simplicidad, no hemos representado gran parte de las jugadas evidentemente perdedores para el segundo jugador. Con las notas n/m, escritas al lado de los diferentes arcos,

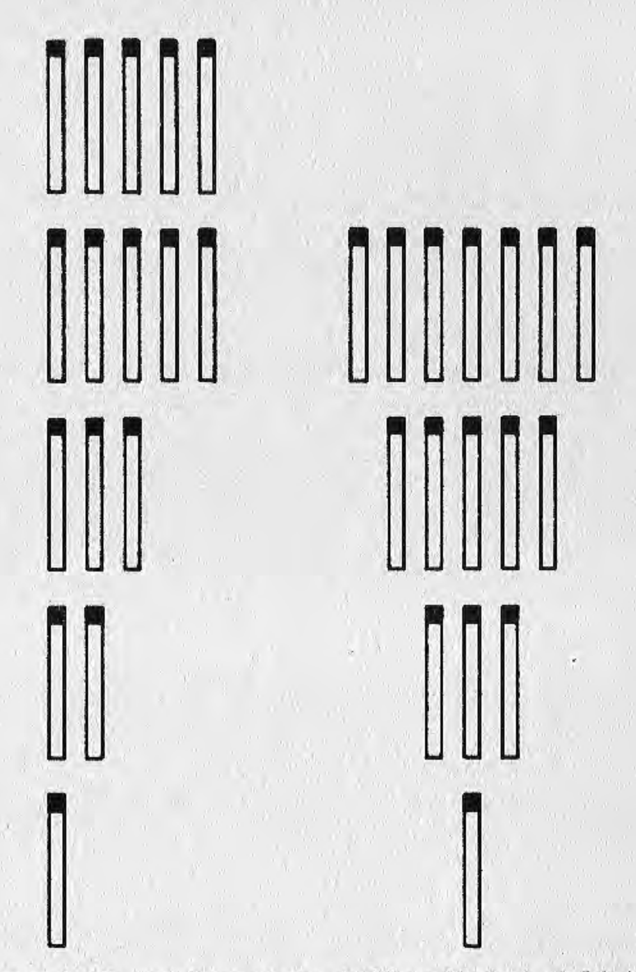


Figura 1.—Dos configuraciones del Nim o Marienbad. A la izquierda la usada en nuestro programa, a la derecha la que se utiliza normalmente para jugar. Las combinaciones posibles son infinitas.

se ha indicado la elección de tomar n objetos de la fila m, mientras que dentro de cada nudo (estado) están representados verticalmente los números residuales de las tres filas. Por razones tipográficas, en el texto que sigue escribiremos, en cambio: 2-2-1, 3-0-1, etc. Finalmente, nótese que los nudos están reagrupados en niveles, J1 y J2 alternativamente, que denotan aquellos en los que la jugada corresponde al primero o al segundo jugador.

Següir el desarrollo del micro-Nim de la figura no es difícil, y menos aún ayudándose, por ejemplo, con cerillas reales. Analizan-

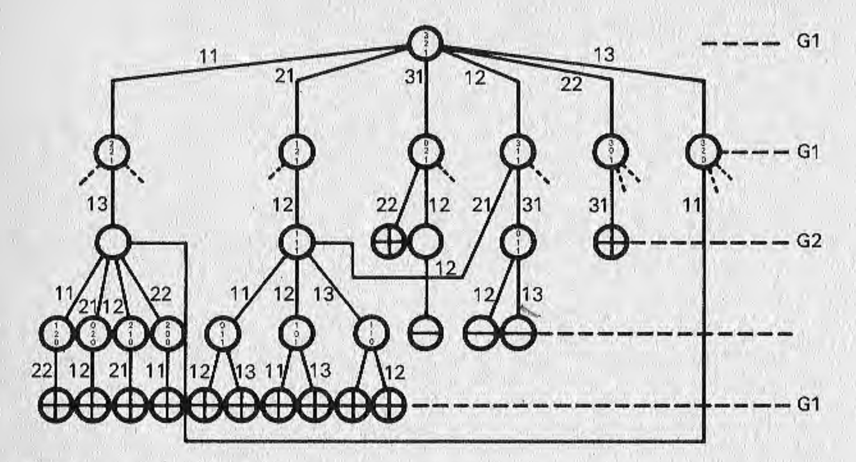


Figura 2.—Arbol simplificado del Marienbad 3-2-1 (a). Para mayor sencillez se han omitido muchas elecciones, especialmente las perdedoras para el segundo jugador J2. Gana "matemáticamente" este último.

do así todas las combinaciones posibles, nos daremos cuenta en seguida que el segundo jugador gana "por narices". Por ejemplo, si después de la jugada 2/1 del primero, el segundo se decide por 1/2 dejando en la mesa 1-1-1, cualquier jugada que haga el otro dejará dos cerillas en filas diferentes: una de estas la cogerá el segundo, obligando al adversario a coger la última, perdiendo la partida.

Para abreviar, observemos que en el grafo (o árbol) de la figura mencionada se indican:

- con trazo grueso las jugadas ganadoras del segundo jugador;
- los nudos finales, caracterizados todos por un estado 1-0-0, 0-1-0 ó 0-0-1, son pequeños círculos que encierran un signo "+" o "–", según venza el segundo o el primer jugador (puede ocurrir esto si el otro se equivoca, por ejemplo, con 1/1 en la configuración 3-2-1, con lo que el primero, si no es tonto, replica con 1/1 y gana);
- partiendo desde abajo, se han colocado oportunamente signos "+" al lado de los nudos que se revelan nudos-raíz de subárboles ganadores para el segundo jugador: subiendo gradualmente es fácil constatar que todos los subárboles que se refieren a la raíz del árbol entero tienen la marca "+", de donde todo el Marienbad 3-2-1 es ganador para el segundo jugador. Nótese también que los nudos 2-2-0 y 1-1-1 se han alcanzado ambos a través de dos caminos diferentes.

Nuestro algoritmo para el Nim

Todo lo que hemos visto en el ejemplo precedente se puede generalizar. En los juegos del tipo que estamos examinando, la victoria es exclusiva de un "predestinado" que, cada vez que le toca jugar a él, sólo tiene que evitar las jugadas que cambiarían la situación, ofreciendo la victoria matemática al adversario. En alguno o en todos los nudos puede suceder que la "jugada feliz" sea una sólamente, pero no importa, es suficiente.

La demostración del teorema ya la hemos visto empíricamente en el párrafo anterior: se basa en la descomposición del árbol del juego en subárboles (empezando por abajo), cada uno de los cuales es ganador para uno de los jugadores y perdedor para el otro. El árbol completo se compondrá —partiendo desde abajo— de subárboles, que salen de la raíz, marcados con "+" o "–", según sea ganador de los respectivos subjuegos el primero o el segundo jugador. Ahora bien, si se llega a una situación como la de la figura 3-a, está claro que, sea cual sea la jugada que haga el primero, estará predestinado a la derrota, salvo generosidad o igno-

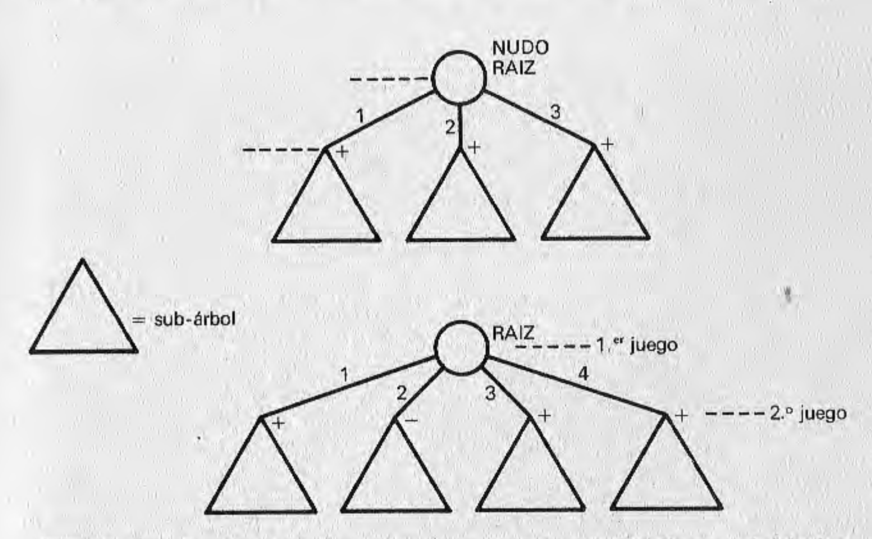


Figura 3.—El árbol de un juego se puede subdividir en subárboles (y sub-juegos), cada uno de los cuales tendrá una combinación ganadora para un solo jugador entre J1 y J2. Si todos los sub-juegos que salen de un único nudo-raíz son combinaciones ganadoras para J2, como sucede en (a), J1 no tiene posibilidad de ganar (excepto si hay algún error por parte de J2). Si, en cambio, hay también un sub-árbol con combinación ganadora para el primer jugador (b), éste se asegura la victoria tomando esa elección (2).

rancia del adversario. Si en cambio, como en b, hay por lo menos un subárbol marcado con "-" es con esta jugada con la que el primero se asegura la victoria.

Volvamos a Marienbad (hoy en día esta ciudad se llama Marianske Lazne...). Es el suyo un juego un poco más serio y amplio que el mísero 3-2-1 visto hasta ahora. Resulta fácil imaginar lo frondoso e intrincado que llegará a ser su árbol correspondiente. Generalmente, los juegos de ordenador se basan en la exploración, más o menos exhaustiva, del subárbol de las jugadas siguientes, pero hacer ésto lateralmente es muy complicado y lento, excepto para los juegos sencillos, así que hay ciertas limitaciones: podaduras, limitaciones de la profundidad, o sea, de los niveles de las jugadas, y otros criterios. De todas formas, el mecanismo de exploración de árboles lo trataremos en el próximo capítulo. Aquí vamos a evitarlo, adoptando otro método, más que nada para eludir el choque repentino con una serie de delicados conceptos informáticos, como las pilas y el backtracking.

Utilizaremos una idea que, por lo menos en este caso específico, simplifica el procedimiento y quizá aumenta la velocidad. Nació por la observación psicológica de como actúa el ser humano cuando juega al Nim o a otros juegos parecidos. Nuestra mente no es capaz de bajar demasiados niveles por lo que, más o menos conscientemente, lo que hacemos es analizar la configuración (en inglés se diría "pattern") consiguiente a cada una de las jugadas posibles en ese nivel; si es reconocida como una de las vencedoras (o como "ventajosa" en los juegos muy complicados) la jugada se efectúa, si no se examina otra (escogida normalmente al azar). Ahora bien, si se tuviera el repertorio de todas las configuraciones ganadoras, bastaría realizar el análisis todas las veces sólo hasta el primer nivel.

Nos explicaremos con un ejemplo. Supongamos que en la mesa tenemos 4-2-1. Pues bien, quien conozca que 3-2-1 es ganadora (para él) no dudará en jugar 1/1. ¿Y si hubiera sido 2-1-5? No se necesita mucho para deducir, por analogía, que 1/3 produce 2-1-3 y que este modelo es totalmente "equivalente" a 3-2-1, así como a 1-2-3 ó 1-3-2. Comprobamos así que no merece la pena acordarse de todas las configuraciones: será suficiente con reconducir las equivalentes a una única común. El criterio es sencillo y se basa en el hecho de que:

- los ceros no cuentan;
- el orden de los datos tampoco cuenta.

Por lo tanto será conveniente hacer referencia a una clasificación preestablecida. Nosotros adoptaremos la decreciente.

Llegados a este punto nos construiremos manualmente una

pequeña base de datos de las configuraciones vencedoras (lo repetimos por última vez: "vencedoras" para quien las deja en la mesa).

El procedimiento para construir estas configuraciones es mecánico y, como veremos, se puede confiar a un programa computerizado. Va desde lo más sencillo hasta lo más complejo (en informática se diría que es de tipo "bottom up"), partiendo por tanto de la configuración más elemental posible, o sea, 1. Como dijimos antes este 1 está en el lugar de todas las posibles (1-0-0-...-0, 0-1-0...-0, 0-0-0-...-1). Limitémonos, por sencillez, a cinco líneas y hagamos girar las cifras de forma que:

- se respete el orden decreciente al representar el estado;
- se hagan crecer, hasta donde sea posible, las líneas inferiores y las líneas vacías;
- después de una configuración m-m-m-...-m se pase a ...-m + 1-0-0-...-0 (que, por lo dicho, se abreviará con ...-m+1, omitiendo los ceros).

En base a estos criterios, después de 3-2-1 viene 3-2-2 mientras que a 4-3-2 le siguen 4-3-2-1 y 4-3-2-1-1. Concretando tendremos:

1; 1-1; 1-1-1; 1-1-1-1; 1-1-1-1-1;
2; 2-1; 2-1-1; 2-1-1-1; 2-1-1-1-1;
2-2; 2-2-1; 2-2-1-1; 2-2-1-1-1; 2-2-2; 2-2-2-1; 2-2-2-1-1; 2-2-2-2;
2-2-2-2-1; 2-2-2-2-2;
3; 3-1; 3-1-1; 3-1-1-1; 3-1-1-1-1;
3-2; 3-2-1; 3-2-1-1; 3-2-1-1-1; 3-2-2; 3-2-2-1; 3-2-2-1-1; 3-2-2-2;
3-2-2-2-1; 3-2-2-2-2;
3-3; 3-3-1; 3-3-1-1; 3-3-1-1-1; 3-3-2; ...etc.

La búsqueda de las configuraciones adecuadas se parece en todo a la de los números primos sucesivos: a cada nueva configuración se le hace la prueba de "invencibilidad", comparándola con las configuraciones vencedoras encontradas hasta ese momento. Si, partiendo de una de las jugadas permitidas, es posible reconducirla a una de las vencedoras, evidentemente se tratará de una configuración vencedora, pero para el adversario.

Por lo tanto se la descarta. En cambio, será incluida la que sale intacta del test, para todas las jugadas posibles. Siguiendo este criterio encontraremos enseguida las primeras configuraciones vencedoras:

1; 1-1-1; 1-1-1-1-1;
2-2; 2-2-1-1; 2-2-2-2;

3-3; 3-2-1

...

Siguiendo con este aburridísimo ejercicio se examinan las configuraciones que siguen, al lado de las cuales se pondrá, entre paréntesis, "VENCEDORA" o, en caso contrario, las configuraciones vencedoras a las que se puede reducir:

3-2-1-1	(3-2-1 ó 2-2-1-1)
3-2-1-1-1	(VENCEDORA)
3-2-2	(3-2-1 ó 2-2)
3-2-2-1	(3-2-1 ó 2-2-1-1)
3-2-2-1-1	(2-2-1-1 ó 3-2-1-1-1)
3-2-2-2	(2-2-2-2)
3-2-2-2-1	(VENCEDORA)
3-2-2-2-2	(2-2-2-2 ó 3-2-2-2-1)
3-3	(VENCEDORA)

....

En los DATA de las líneas 4 a 12 del programa de la figura 4 se representan posteriores configuraciones Nim vencedoras. Quien lo desee puede intentar encontrar otras. Cuantas más combinaciones ganadoras memoricemos más posibilidades tendremos de ganar fácilmente a los ignorantes y desprevenidos.

```
2 REM LE PASADO AÑO EN MARIENBAD (ALIAS "NIM")
4 DATA " 1"," 1 1 1"," 1 1 1 1 1"," 2 2"," 2 2 1 1"," 2 2 2 2"
6 DATA " 3 2 1"," 3 2 1 1 1"," 3 2 2 2 1"," 3 3"," 3 3 1 1",
  " 3 3 2 2"
8 DATA " 3 3 3 2 1"," 3 3 3 3"," 4 4"," 4 4 1 1"," 4 4 2 2",
  " 4 4 3 2 1"," 4 4 3 3"
10 DATA " 4 4 4 4"," 5 4 1"," 5 4 1 1 1"," 5 4 2 2 1"," 5 4 3 2",
  " 5 4 3 2 1"
12 DATA " 5 4 4 4 1","5 5"," 5 5 1 1"," 5 5 2 2"
14 DATA 5,5,3,2,1:REM DATOS DE CONFIGURACION INICIAL
100 LIM=29:NL=5:DIM CV$(LIM),S(NL):VERDADERO=1:FALSO=0
110 FOR I=1 TO LIM:READ CV$(I):NEXT I
120 FOR R=1 TO NL:READ S(R):NEXT R
130 HOME:PRINT "HAY VARIAS FILAS DE CERILLAS":PRINT
140 PRINT "   FILA NUMERO","   CERILLAS   ":PRINT
150 FOR I=1 TO NL:PRINT "          ";I,"          ";S(I):NEXT I
160 PRINT:INPUT "ESCOJE LINEA:";L:IF L<1 OR L>NL THEN 160
170 INPUT "ESCOJE CERILLAS : ";NC:IF NC<1 OR NC>S(L) THEN 160
180 S(L)=S(L)-NC
190 REM EL ORDENADOR ESTUDIA LA JUGADA
200 GOSUB 500:REM COPIA S EN LA MATRIZ-TRABAJO W
210 FOR I=1 TO NL
220 IF S(I)=0 THEN 360:REM VE A NEXT SALTANDO LINEA VACIA
230 FOR PRUEBA=1 TO S(I)
240 W(I)=W(I)-PRUEBA
250 GOSUB 1000:REM DOUBBLE-SORT DE W (NUEVO)
260 GOSUB 1200:REM TRANSF.DE W EN CADENA C$
270 REM BUSQUEDA DE C$ EN MATRIZ CV$
280 ENCU=FALSO
290 FOR K=1 TO LIM
300 IF C$=CV$(K) THEN ENCU=VERDADERO
310 NEXT K
320 IF ENCU THEN NC=PRUEBA:LINEA=I:PRUEBA=S(I):GOTO 340
330 GOSUB 500:REM RESTABLECE S EN W
340 NEXT PRUEBA
350 IF ENCU THEN I=NL
360 NEXT I
370 IF NOT ENCU THEN PRINT "ERROR EN LOS DATAS":STOP
380 PRINT:PRINT "YO,ORDENADOR,QUITO";NC;"CERILLAS"
390 PRINT "DE LA LINEA ";LINEA
400 S(LINEA)=S(LINEA)-NC:FOR K=1 TO 1000:NEXT K
410 IF C$<>" 1" THEN 140
420 PRINT:PRINT "...QUEDANDO EN LA MESA"
430 PRINT "UNA SOLA CERILLA,HE GANADO!"
440 PRINT:INPUT "OTRA PARTIDA? (S/N)";R$
450 IF LEFT$(R$,1)="S" OR LEFT$(R$,1)="s" THEN RESTORE:GOTO 110
460 END
490 REM COPIA DE S EN W
500 FOR K=1 TO NL:W(K)=S(K):NEXT K:RETURN
990 FOR K=2 TO NL
1000 IF W(K)>W(K-1) THEN C=W(K): W(K)=W(K-1):W(K-1)=C:K=1
1020 NEXT K:RETURN
1190 REM TRANSFORMACION MATRIZ W EN CADENA C$
1210 FOR K=1 TO NL
1220 IF W(K)=0 THEN K=NL:GOTO 1240:REM OMITE CEROS
1230 C$=C$+" "+STR$(W(K))
1240 NEXT K:RETURN
```


Figura 4.—Listado del programa que siempre gana el Marienbad 5-5-3-2-1 (realizado en el BASIC del Apple).

El programa

Una vez definido el algoritmo y, con él, su respectiva máquina de estados variables arbórea (nótese que esta vez el algoritmo la sobreentiende y, en cierto sentido, la sobrepasa) queda por me-

ditar una estructura de datos apropiada. Después de ello hacer un programa correctamente será sólo cuestión de paciencia, aunque necesitará bastante, porque incluso un pequeño fallo (olvidar el RETURN en una subrutina, equivocar las modalidades ejecutivas de una instrucción BASIC, etc.) será suficiente para hacer tambalearse al más comprobado de los algoritmos. Al final podremos asombrar (moderadamente) a nuestros amigos poniéndolos delante de una máquina que siempre gana.

Sin pretender que sea la solución óptima, sugerimos la matriz y la cadena como estructuras de datos apropiadas. En efecto, nos parece que:

- la matriz (de cinco elementos en nuestro caso) es cómoda para realizar la clasificación (descendente) y las diferentes "descarnaduras" de elementos;
- la cadena resulta una representación sintética de las configuraciones, especialmente con la finalidad de una comparación inmediata (sobre la ocupación de espacio y la extensión del método a juegos Nim más amplios habría mucho que discutir).

No nos queda pues más que diseccionar, trozo por trozo, el programa propuesto. Generalmente, hemos reducido todo a sus términos esenciales, con comentarios muy escuetos e interface-usuario espartano. Quien tenga ganas podrá, por ejemplo, usar los gráficos (para representar visualmente las cerillas) y otras facilidades difícilmente transportables de un micro a otro.

Los DATA iniciales representan las configuraciones ganadoras. Hemos encontrado más sencillo sustituir el trazo separador por el espacio en blanco (en inglés blank), lo que asegura una buena transferibilidad entre diferentes dialectos BASIC. De todas formas hay que tener cuidado y hacer preceder la cadena por un espacio. Esto se ha hecho en atención al BASIC Microsoft y derivados (por ejemplo el del Commodore 64) en los que la función STR$(X), pone, sistemáticamente, un espacio delante del número X, al convertirlo en cadena (o sea, transforma 1234 en " 1234"). El programa de la figura, para alternar dialectos de similar popularidad, se refiere al BASIC del Apple. De todas formas, las modificaciones que hay que hacer son mínimas. Los microsoft-istas sólo tendrán que cambiar, en la línea 110, VERDADERO=1 por VERDADERO=-1 y la línea 1230 así:

```
1230 C$=C$+STR$ (W(K))
```

Volviendo al listado (línea 110), las configuraciones ganadoras se cargan enseguida en la matriz CV$. Inmediatamente se po-

nen en las seis casillas del vector de estado (S) los valores 5, 5, 3, 2, 1, que constituyen el Nim de calibre medio gracias al cual el ordenador, jugando como segundo, nos gana incansablemente.

Desde la línea 130 a la 160 la máquina propone al adversario humano que mueva, después de mostrarle la situación inicial con el cuadro siguiente:

NUMERO DE FILAS	CERILLAS
1	5
2	5
3	3
4	2
5	1

Un detalle: en la línea 170 el control IF NC<1 OR NC>E(L), cuya misión es impedir que el usuario haga trampa eligiendo cero cerillas o un número mayor de las presentes, NO se cierra con THEN 170, como se podría esperar. El THEN 160, envía a la elección de otra línea y sirve para el caso de que E(L)=0. Si se hubiera hecho THEN 170 estaríamos atrapados en un bucle sin fin.

Después de que el ordenador ha tomado nota de nuestra jugada (línea 180), quitando las NC Cerillas de la línea L elegida, se inicia el estudio de la jugada. Para mayor comodidad, se citan aquí las líneas correspondientes:

```
190 REM ESTUDIO DE LA JUGADA
    POR PARTE DEL ORDENADOR
200 GOSUB 500:REM COPIA DE E EN
    LA MATRIZ DE TRABAJO W
210 FOR I=1 TO NL
220 IF E(I)=0 THEN 360:REM VETE
    A NEXT,SALTANDO LA LINEA VACIA
230 FOR PRUEBA=1 TO E(I)
240 W(I)=W(I)-PRUEBA
250 GOSUB 1000:REM ORDENACION POR
    BURBUJA DE W (NUEVA)
260 GOSUB 1200:REM TRANSFORMA W
    EN CADENA C$
270 REM BUSQUEDA DE C$ EN MATRIZ CV$
```

```
280 ENCU=0
290 FOR K=1 TO LIM
300 IF C$=CV$(K) THEN ENCU=VERDADERO
310 NEXT K
320 IF ENCU THEN NC=PRUEBA:LINEA=I:
    PRUEBA=E(I):GOTO 340
330 GOSUB 500:REM RESTABLECE E (NO ENCONTRADO)
    EN W
340 NEXT PRUEBA
350 IF ENCU THEN I=NL
360 NEXT I
```

Para que resulte más evidente se ha utilizado en el listado una indentación, que aisla mejor los ciclos más internos, con indices PRUEBA y K, "anidados" dentro del bucle principal, de indice I.

El ordenador, después de haber copiado E en W (en la subrutina 500) prueba a quitar, línea por línea, un número de cerillas creciente PRUEBA desde 1 hasta el número E(I) presente en la línea I. Entonces hace la ordenación de W y la transforma en una cadena C$ que va a buscar en la base de datos CV$ (líneas 280-310). Antes de pasar al siguiente ciclo de PRUEBA (¡cuidado!) es necesario restablecer (con la subrutina 500, llamada en la línea 330) la matriz originaria E en W, ya que la matriz de trabajo ha sido manipulada. Cuando por fin se sale, con el NEXT de la línea 360, sabemos que el booleano ENCU tiene que estar activado. La prudencia nos aconseja añadir la siguiente línea:

```
370 IF NOT ENCU THEN PRINT "¿ERROR EN LOS DATA?":STOP
```

En efecto, un descuido o un error de transcripción, siempre es posible en una fase de ampliación del juego o de escritura de los DATA. Como alternativa, quien lo desee puede dejar, aposta, los DATA incompletos (sería como dejar parcialmente ignorante al ordenador) e insertar en la línea 370 un adecuado GOSUB, que nos mande a una rutina en la que el ordenador elija la jugada con criterios aleatorios (naturalmente, después será necesario modificar un poco el programa para prever que el ordenador admita la victoria del hombre).

Ahora, el resto del programa y las distintas subrutinas, ya deberían ser lo suficientemente auto-explicativas y su examen es más útil que cualquier intento de explicación.

Unicamente, y con respecto a los ciclos anidados, damos un consejo. Es una buena regla evitar, excepto en los casos más sencillos, salir fuera de un bucle directamente: en muchos BASIC nos

toparíamos con una interrupción y la señalización de error "NEXT WITHOUT FOR" en el primer encuentro con un NEXT de otro bucle externo. Será más seguro forzar el valor final antes de hacer un GOTO a la línea que contenga el NEXT de aquel ciclo. Consulte para esto las líneas 300 y 320.

El generador de Nim-pattern

Redactar un programa que nos ofrezca las combinaciones ganadoras del Marienbad sería un excelente ejercicio mental. La solución que proponemos aparece en la figura 5. Dado que todos los mecanismos ya han sido suficientemente explicados, los más adelantados deberían intentar soluciones de propia cosecha. Los más listos y voluntariosos podrán, incluso, fundir los dos progra-

```
40 REM ****************************************
50 REM * GENERADOR DE CONFIGURACIONES GANADORAS *
60 REM *          DEL JUEGO DEL "NIM"           *
70 REM *      (TAMBIEM PROGRAMA MARIENBAD)      *
80 REM ****************************************
90 NC=4:NL=5:DIM CV$(200),S(NL),W(NL):VERDADERO=1
100 CNC=1:S(0)=NC:S(1)=1:C$=" 1":REM PRIMERA CADENA GANADORA
110 GOSUB 400:REM CARGADO EN CV$
120 REM CREACION NUEVA CONFIGURACION
130 M=2:REM MARCHA HACIA ADELANTE
140 FOR X=M TO NL
150 S(X)=S(X)+1
160 GOSUB 500:REM PRUEBA DE INVENCIBILIDAD
170 IF NOT ENCU THEN GOSUB 300:GOSUB 400
    REM CARGA CONFIG.SI NO REDUCIBLE
180 NEXTX
190 IF S(NL)=NC THEN 240:REM PRINT FINAL
200 REM MARCHA ATRAS
210 FOR X=NL TO 1 STEP -1
220 IF S(X)=S(X-1) THEN S(X)=0:NEXT X
230 M=X:GOTO 140
240 REM PRINT FINAL DE LAS CV$
250 FOR K=1 TO CNC:PRINT CV$(K);" ";:NEXT K
260 END
300 REM TRASFORM.MATRIZ-TRAB EN CADENA
310 C$=""
320 FOR K=1 TO NL
330 IF W(K)=0 THEN K=NR:GOTO 350
```

```
240 C$=C$+" "+STR$(W(K))
350 NEXT K:RETURN
400 REM CARGA EN MATRIZ CV$
410 CV$(CNC)=C$:CNC=CNC+1
420 RETURN
500 REM PRUEBA DE INVENCIBILIDAD
510 GOSUB 700:REM COPIA S EN MATRIZ-TRAB. W
520 FOR I=1 TO NL:IF S(I)=0 THEN I=NL:GOTO 640
530 FOR PRUEBA=1 TO S(I)
540 W(I)=W(I)-PRUEBA
550 GOSUB 750:GOSUB 300:REM SORT+TRANSF.EN CADENA C$
560 ENCU=0:REM BUSQUEDA DE C$ EN MATRIZ CV$
570 FOR K=1 TO CNC-1
580 IF C$=CV$(K) THEN ENCU=VERDADERO:K=CNC-1
590 NEXT K
600 IF ENCU THEN PRUEBA=S(I):GOTO 620
610 GOSUB 700:REM RESTABLECE S EN W
620 NEXT PRUEBA
630 IF ENCU THEN I=NL
640 NEXT I
650 RETURN
700 REM COPIA S EN MATRIZ-TRAB. W
710 FOR K= 1 TO N:W(K)=S(K):NEXT K:RETURN
720 REM DUBBLE-SORT (ORDEN DECREC.) DE W
760 FOR K=1 TO NL-1
770 IF W(K)<W(K+1) THEN C=W(K):W(K)=W(K+1):W(K+1)=C:K=0
780 NEXT K:RETURN
790 REM FIN DE PROGRAMA
```

Figura 5.—Programa que origina las combinaciones ganadoras. Las subrutinas comunes con el anterior tienen aquí números de línea diferentes.

mas, con algún menú inicial, teniendo en cuenta que ambos se basan en el test de "invencibilidad".

Con respecto al programa anterior, el que vamos a ver de inmediato tiene como novedad el mecanismo de la generación de configuraciones (ganadoras y no ganadoras). La solución propuesta figura en las líneas 120 a 130 (NC y NL son los números de columna y línea elegidas). Hay que tener en cuenta que los números de la subrutinas no se corresponden con las del programa precedente:

```
120 REM CREACION NUEVA CONFIGURACION
130 M=2:REM MARCHA ADELANTE
```



```
140 FOR X= M TO NL
150 S(X)=S(X)+1
160 GOSUB 500:REM PRUEBA DE "INVENCIBILIDAD"
170 IF NOT ENCU THEN GOSUB 300:GOSUB 400
180 NEXT X
190 IF S(NL)=NC THEN 240:REM PRINT FINAL
200 REM MARCHA ATRAS
210 FOR X=NL TO 1 STEP -1
220 IF S(X)=S(X-1) THEN S(X)=0:NEXT X
230 M=X:GOTO 140
```

El primer truco consiste en poner NC (o sea, el número máximo de cerillas por línea) en el elemento de índice O,S(0) (ver figura 6). Al principio, una vez cargada la cadena C$="1" en CV$(1) y establecido S(1)=1 —mientras que de S(2) a S(NL) se tendrán todos ceros— se ejecuta la "marcha adelante", como hemos llamado a la primera fase. Esta consiste en incrementar en 1 todos los elementos, desde M=2 en adelante (línea 140). Así, al primer paso tendremos: "1"; "111";...; "11111". A continuación, M tomará otros valores, así que de una configuración inicial como "32111" se pasará (con M inicial igual a 3) a: "32211"; "32221"; "32222".

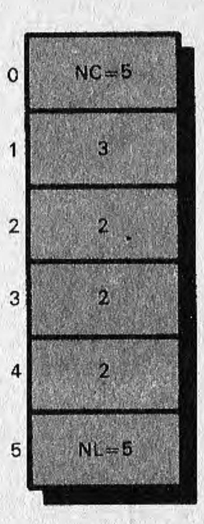


Figura 6.—Matriz de estado S. En la generación de las sucesivas configuraciones se pone S(0)=5. Así, en la fase hacia atrás de puesta a cero el test IF S(X)=S(X-1) proporciona para X=1, S(1) <>S(Q) y el 3 de la figura no se convierte en un cero (ver texto).

¿Qué ocurre en la otra fase, llamada "marcha atrás" (líneas 200-230)? Es muy simple: partiendo desde abajo (X=NL) hasta arriba, se ponen a cero todos los S(X) que coincidan, parándose en el primero que resulte ser de valor mayor. De este modo, por ejemplo, a "43222" le seguirá "43000" (como cadena tendremos en realidad, "43"), mientras que a "32222" le seguirá "3". Y esto, gracias al truco de haber establecido S(0)=NC. Ya no queda más que hacer (línea 230) M=X y retomar la marcha adelante, hasta que S(NL)=NC; no se repite la marcha atrás, sino que se llega a la impresión final. Si quiere puede comprobar que el proceso recorre todas las configuraciones (ganadoras y perdedoras). Basta para ello con hacer un par de cambios:

- borrar la línea 160;
- modificar la línea 170 así:

```
170 GOSUB 700: GOSUB 300: GOSUB 400
```

(después de haber copiado cada S en W lo convierte en cadena y lo carga en CV$). El programa, tras el RUN, hará una lista de todas las configuraciones, empleando mucho menos tiempo que el programa completo. Este último, con todas sus búsquedas, pruebas y ordenaciones por burbujas, se muestra bastante renqueante apenas sobrepasa los valores NC=NL=5. Con NL=5 y NC=6 el Apple II tarda doce minutos para generar las 59 CV$ ganadoras.

Se podrían obtener tiempos sensiblemente menores aplicando, para la búsqueda en las tablas, la técnica de la bisección, que consiste en buscar primero en una de las mitades de la matriz, después en la mitad de la mitad y así sucesivamente. Quien la conozca, nos objetará que esta técnica requiere que CV$ esté ordenado. Pues bien, lo está, a pesar de las apariencias que pueda dar la extraña generación de los elementos sucesivos. En efecto, se trata de cadenas y no de números, y en la clasificación de éstas (son datos de tipo alfabético) una cadena como "44" es considerada mayor que una, como "4332211", aunque sea más larga (lo que confirma lo acertado del proceso de generación y de la elección del tipo cadena, ¿no les parece?).

Pero la corrección que les sugerimos es todavía más sencilla: se basa en la consideración de que, cada vez que se genera una nueva CV$, es probable que pudiera ser "absorbida" por una CV$ generada recientemente (ejemplo: "44221" por "4422"); las primeras CV$ ganadoras serán evocadas cada vez menos.

La experiencia confirma esta intuición (razonada), pues los tiempos se ven prácticamente reducidos a la mitad. La modificación (nos olvidábamos) es esta:


```
570 FOR K=CNF-1 TO 1 STEP -1
580 IF C$=CV$(K) THEN ENCU=VERDADERO:K=1
590 NEXT K
```

Una vez más el software se muestra como una materia en continua evolución.

CAPITULO IV

ESTRATEGIAS INTELIGENTES

Recordando un antiguo problema

Veremos ahora un problema de estrategia computerizada sencillo, como podrá apreciar, pero delicadísimo. El ordenador deberá encontrar la mejor solución para transportar un lobo, una cabra y una coliflor, desde una orilla de un río hasta la otra, disponiendo de una barca con capacidad para un solo pasajero por viaje (aparte del barquero, claro) y evitando que el lobo se coma la cabra y la cabra la coliflor.

El árbol de este juego se construye sin muchas dificultades. También será inevitable un barrido exhaustivo. Observando la figura 1 notará que el árbol aparece bastante podado pero, paradójicamente, tiene una amplitud infinita. Para comprender su evolución, no debemos avergonzarnos si utilizamos tres carteles con los rótulos "l", "cb" y "co" —como figuran en el grafo para los tres compañeros del campesino— para que nos guíen en sus posiciones y movimientos entre las dos orillas. El árbol de la figura se entenderá mejor si tiene en cuenta que:

- los nudos están señalados alternativamente con D e I (por orilla Derecha y orilla Izquierda: la orilla Derecha se supone que es la de partida);
- las ramas de elección aparecen señaladas con los carteles ya vistos ("l", "cb", "co") además de "n", que significa que no se transporta a nadie;
- el signo + indica el éxito del juego (todos los protagonistas están en la otra orilla);
- los nudos marcados con "—" suponen jugadas perdedoras.

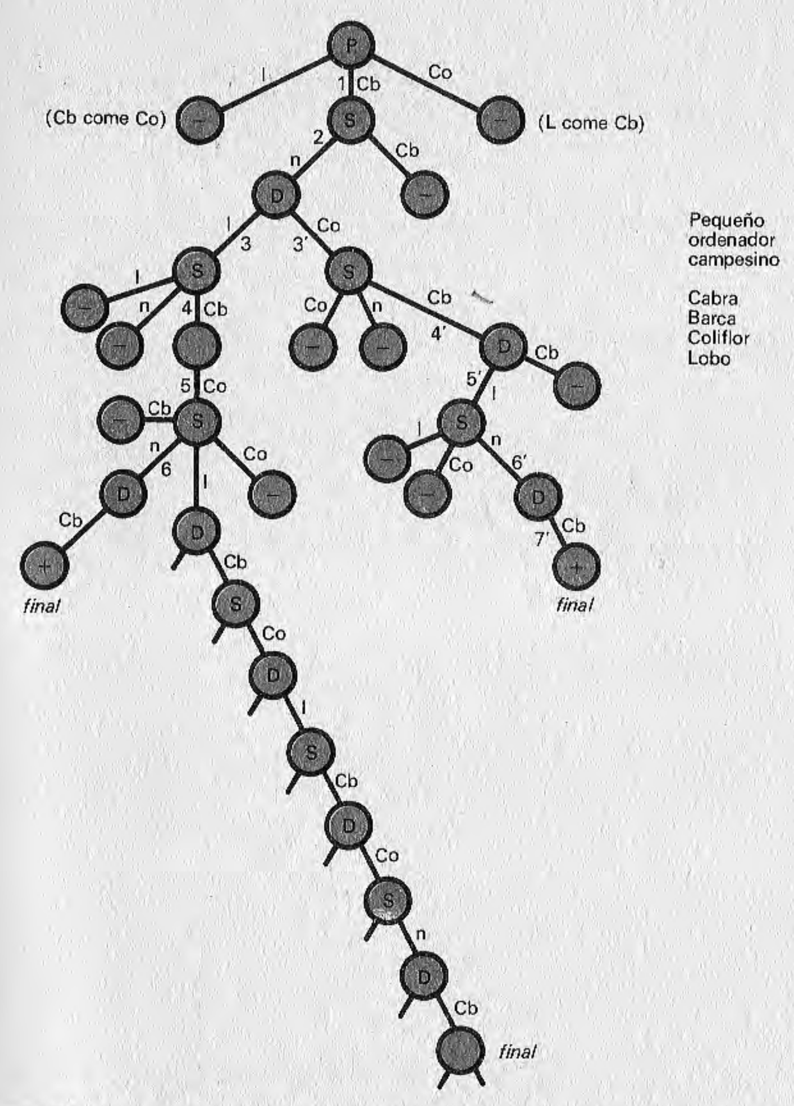


Figura 1.—Arbol simplificado del juego del lobo, la cabra, y la coliflor. Se advierten dos estrategias abreviadas e infinitas soluciones "degeneradas", no optimizadas.

Con el fin de que el árbol quede podado al máximo, no sólo se han excluído elecciones que supongan que el lobo se come a la cabra o ésta a la coliflor, sino también las que en la vuelta transporten lo mismo que se llevó a la ida. En la orilla izquierda, como se verá, se ha hecho algo más específico.

El análisis de la figura revela que las estrategias vencedoras optimizadas son dos, y se consiguen con los siguientes movimientos en los dos sentidos:

- cabra (primera jugada obligada) - nadie - lobo - cabra (vuelta) - coliflor - nadie - cabra;
- cabra - nadie - coliflor - cabra - lobo - nadie - cabra.

Se trata de dos grupos de siete viajes señalados en la figura con 1,2,3,4,5,6,7 y con 1,2,3',4',5',6',7'.

Precisemos ahora la tarea que se quiere confiar al programa. En esencia esta consiste en: encontrar la jugada sucesiva simulando el desarrollo de la máquina de estados variables y justificar al operador las jugadas descartadas.

Siguiendo los criterios tantas veces discutidos, lo más importante es, ante todo, definir una estructura de datos adecuada. Un primer ejemplo, para que el lector piense en él, aparece en la figura 2, en lenguaje Pascal, junto a dos de los procedimientos posibles. En particular se apreciarán los tipos estructurados siguientes:

```
TYPE pasaj=[nadie, lobo, cabra, coliflor]
        orilla=SET OF pasaj;
```

Recordamos, tanto a los pascalistas como a los que no lo son, que un tipo SET está formado por todas las combinaciones posibles —desde ninguno a todos— de los elementos del tipo-dato que sigue a OF. En este caso serían: el conjunto vacío "lobo"; "lobo y cabra" y sucesivamente hasta llegar a "lobo, cabra y coliflor". Aplicadas a las variables "ordech" y "orizq", precisarán con claridad la situación en las orillas y facilitarán mucho el planteamiento de los procedimientos. Por ejemplo, el (hipotético) procedimiento "valcomidech" (por "valoración de la comilona en la orilla derecha") incluiría la frase:

```
IF lobo IN ordech AND cabra IN ordech
THEN comil=lobo; victima:=cabra
```

que posee una gran elocuencia incluso para quien no sepa que IN hace el test de pertenencia del lobo o la cabra a la variable de tipo SET ordech.

En la misma se usa la estructura en pila (stack). La pila servirá también para el fin primario, didáctico, de entender como esta estructura, llamada a veces LIFO (Last In First Out: último en en-

```
PROGRAM cabracol;
TYPE pasaj=[nadie,lobo,cabra,coliflor]
     orilla=SET OF pasaj;
VAR ordech,orizq:orilla;
    pas,comil,victima:pasaj;
    numdech,nimizq:INTEGER;
    primvuel,dir:comil:BOOLEAN;
    pila:matriz[1..200] OF pasaj;
    dirpila:INTEGER;
PROCEDURE inicialic;
  BEGIN
   ordech:=[lobo,cabra,coliflor];
   orizq:=[]
   dir:=FALSE;
   pasaj:=nadie;
   primvuel:=TRUE;
(* etc *)
  END;
PROCEDURE valdech(* valor "comilona" en orilla dech *)
  BEGIN
   comilona:=TRUE;
   IF numdech<>2 THEN comilona:=false;
    ELSE IF lobo IN ordech AND cabra IN ordech
            THEN comil:=lobo;victima:=cabra;
            ELSE IF cabra IN ordech AND
                    coliflor IN ordech
                 THEN comil:=cabra;victima:=coliflor;
 END;
(* resto de programa *)
```

Figura 2.—Posible implementación (a título indicativo meramente) de la estructura de datos del juego anterior en Pascal, con el bosquejo de dos procedimientos que los utilizan, caracterizados por la autoexplicación típica de este lenguaje.

trar, primero en salir) funciona para el barrido de árboles de juego y distintos laberintos. Precisamente la pila sirve para:

- amontonar las elecciones hechas una sobre otra y, esto también es importante, para conservar el estado del sistema anterior a la jugada actual;

- recuperar un estado precedente (de nivel inferior) en el caso que en un nudo de un árbol TODAS las jugadas se hayan agotado, después de lo cual, vueltos a un nivel anterior, podremos empezar "desde el punto en el que nos hubiéramos quedado" a estudiar una jugada distinta a la que nos llevó a un callejón sin salida.

Esta es la esencia del mecanismo llamado backtraking (vuelta sobre nuestros propios pasos) que, en lenguajes como LISP o Prolog está disponible en forma transparente para el usuario. Para mayor claridad, esto está ligado al concepto de recursividad, sobre el que no insistiremos, aunque les recomendamos repasar el capítulo correspondiente de "Introducción al Pascal: programación estructurada".

En cuanto a la pila (debe su nombre a la posibilidad de ser asimilada a una pila de platos) crece o decrece a través de operaciones opuestas llamadas PUSH y POP respectivamente. Observe la pila ilustrada en la figura 3 y, seguidamente, la figura 7, que ilustra como se usará la pila en el caso de nuestro transbordo estratégico.

Estructura de datos necesaria en BASIC

Está ilustrada en la figura 4. Al lado de las variables particulares de tipo booleano (lógico, on/off, desviadores o como se las quiera llamar) y a la pila de cadenas STK$, de las que hablaremos a su debido tiempo, debemos destacar:

- la matriz AN$ que tiene cuatro elementos (incluída "nadie") y que usaremos en los mensajes que indiquen el nombre del pasajero a bordo de la barca;
- las matrices OD y OI que se refieren a la situación en las orillas derecha e izquierda; el elemento de índice nulo indica el número de personajes allí presentes (excluído el campesino), mientras que los otros elementos pueden valer 0 ó 1, denotando la ausencia o presencia de su elemento asociado, en el orden siguiente: lobo, cabra, coliflor. Así, por ejemplo, el estado OD(0)=2; OD(1)=1; OD(2)=0; OD(3)=1 indica lobo y coliflor en la derecha;
- las variables PAS, PD y PI, cuyo rango está entre 0 y 3 (0="nadie"; 1="lobo", etc.) son, respectivamente, el PASajero actual de la barca y los índices usados para escoger al nuevo pasajero en las dos orillas;
- DIR es un booleano que indica la DIRección actual: de derecha a izquierda o viceversa.

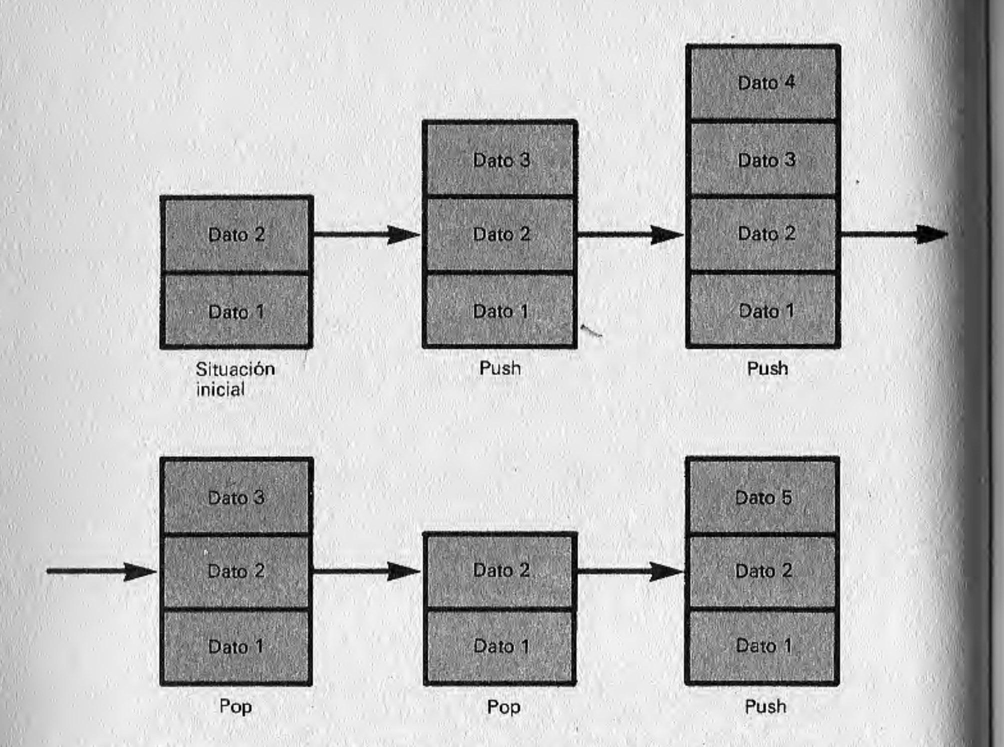


Figura 3.—La pila (stack) se puede asimilar con un montón de platos. La operación PUSH (empuja) coloca un "plato" nuevo en su cima (carga datos), mientras que la operación contraria, POP, extrae los platos (datos) en un orden inverso al de apilamiento (el último que se pone es el primero que se toma). La figura ilustra hipotéticas operaciones suvesivas de PUSH y POP.

Sobre la posible redundancia de los datos se puede discutir mucho, pero nosotros reivindicamos la virtud de la claridad y del "más vale abundar... " tanto aquí como en otros aspectos de la vida.

En la figura 5 aparece el diagrama de flujo del conjunto. Se aprecia que a la fase de inicialización de los datos sigue una fase de "polling de usuario" (polling equivale, más o menos, a investigación) en la que el ordenador precisa, a requerimiento nuestro, lo que está haciendo. La instrucción elemental DIR=NOT DIR invierte a cada ciclo —de 0 a 1 y viceversa— el indicador de ruta. De esta forma, un poco más adelante, se tiene la posibilidad de escoger o alternar, en los giros pares e impares, la jugada referente a la orilla derecha o izquierda. Como veremos en breve, no ha sido posible unificar en un solo procedimiento los dos subprogramas, aunque son bastante parecidos.

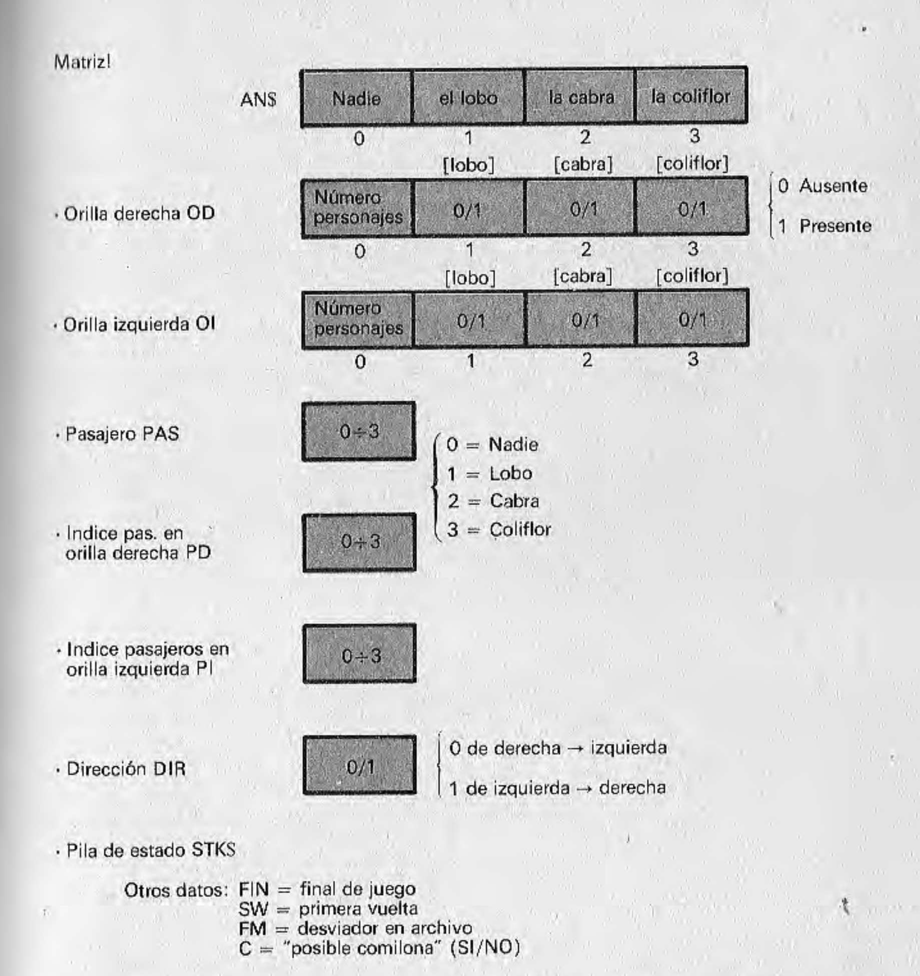
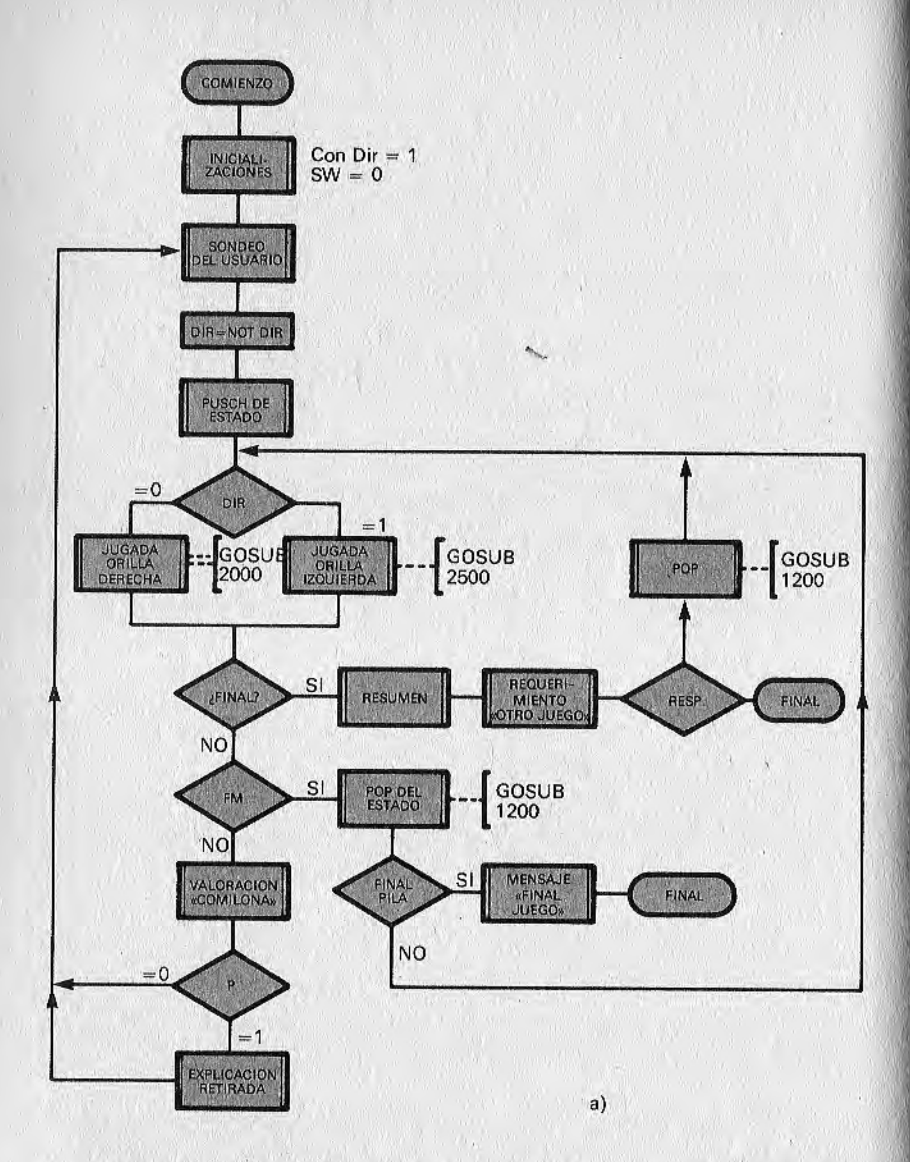


Figura 4.—Estructura de datos en BASIC. Se le puede criticar una cierta redundancia, pero esto favorece la claridad de los planteamientos.

Antes de tales valoraciones se dispone de la subrutina PUSH, para guardar el estado actual en la pila (o bien, anticipando cosas, las matrices AN$, OD, OI y las variables PAS, PD, PI, etc.). Supongamos ahora que FIN y FM son dos booleanos que representan, el primero, el traslado completo del trío y el segundo el Final de los Movimientos del nivel. En el caso de una respuesta afirmativa a la pregunta ¿FINAL? iremos al epílogo y conclusión del juego (lo veremos), mientras que con FM apagado se vuelve a empezar



con el bucle principal. En cambio ¿qué pasa cuando FM está "on"? Está claro: agotadas las posibilidades de jugada en ese nivel, se ejecuta POP para subir de nivel. Para aclarar este asunto recuerden el árbol de la figura 1; imagínense que se encuentran en el nudo de tipo D siguiente a la jugada número 6 y que (hipótesis absurda) todas las jugadas que se han intentado han dado un resultado negativo: el POP deberá llevarles al nudo S anterior.

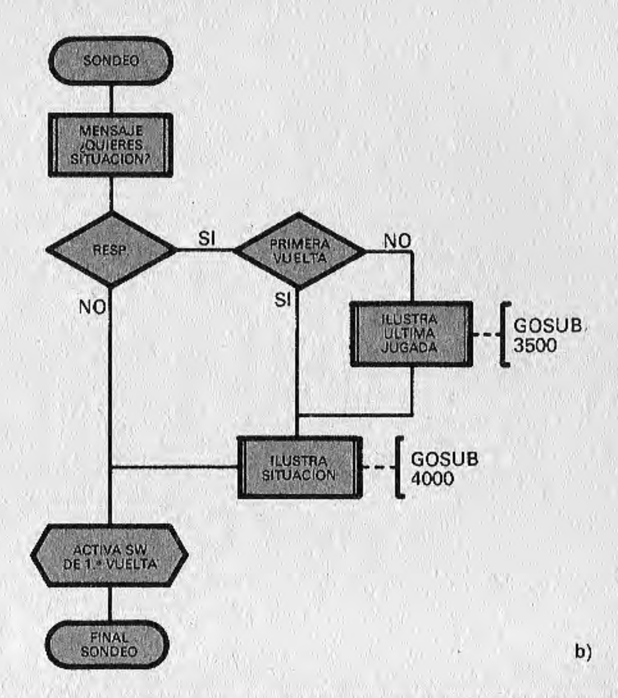


Figura 5. a) —Esquema general del programa. b) Detalle del subprograma "sondeo del usuario".

Siguiendo con el flujo, después se examina la pila: si estuviese completamente vacía significaría que todas las posibles calles han sido exploradas y el algoritmo acaba con un mensaje similar a "FINAL JUEGO". En caso contrario nos trasladamos al punto X, debajo del PUSH, para entrar en la jugada siguiente (el transbordo del lobo en la hipótesis que hicimos). Cuando, en cambio, se alcanza la condición FINAL, se hace el RESUMEN seguido del requerimiento "¿OTRA VUELTA?" Si el usuario contesta que sí, se ejecuta un POP posterior, la vía más directa y lógica para explorar una solución alternativa a la recién encontrada.

El listado y las explicaciones

Resumiendo, el programa está planteado de forma que el ordenador, con las reglas del juego en su memoria, sea capaz de barrer de modo exhaustivo el árbol entero, señalando todas las posibilidades y terminando automáticamente cuando haya recorrido todo el camino y encontrado todas las soluciones. Es impor-

tante subrayar que, a priori, no podemos excluir que NO haya ninguna solución; en tal caso se saldrá respondiendo con "SI" a la pregunta ¿"FINAL DE JUEGO"? sin que la condición FINAL y el sucesivo RESUMEN hayan tenido lugar. En suma, el programa resuelve "mecánicamente" el problema (deberán "olvidarse" de que ya conocen —por lo menos— dos soluciones).

El listado del programa, en BASIC, está representado en la figura 6 y refleja el diagrama de flujo del que ya hemos hablado, excepto por la instrucción DIR=NOT DIR, que no está colocada exactamente como en el diagrama. Hay que precisar que el programa ha sido escrito para un ordenador personal Apple IIe, por lo cual harán falta pequeñas modificaciones del tipo de CLS en vez de HOME y, sobre todo, revisar algunos tratamientos booleanos en los dialectos en los que "verdadero" esté codificado con "-1" en lugar de con "1" como en el BASIC Applesoft.

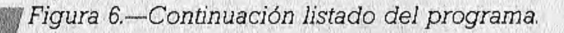
```
10 REM %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
20 REM %%%    SALVAR CABRA   COLIFLOR       %%%
30 REM %%%     JUEGO DE MICRO I.A.          %%%
40 REM %%%                                  %%%
50 REM %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
60 HOME
70 REM INICIALIZACIONES
80 AN$(0)="NADIE"
90 AN$(1)="EL LOBO"
100 AN$(2)="LA CABRA"
110 AN$(3)="LA COLIFLOR"
120 REM PREPARACION ESTADO ORILLA
130 OD(0)=3
140 FOR I=1 TO 3:OD(I)=1:NEXT I
150 OI(0)=0
160 FOR I=1 TO 3:OI(I)=1:NEXT I
170 DIR=0::PAS=0:SW=0:FIN=0
180 PD=0:PI=0
190 REM
200 REM PSEUDOPILA DE ESTADO
210 DIM STK$(200):REM EX ABUNDANTIAM!!
220 IS=0:REM INDICE PILA
230 REM
```

Figura 6.—Listado del programa.


```
240 REM SONDEO USUARIO
250 PRINT "QUIERES LA SITUACION?"
260 PRINT "(<S>=SI,<OTRO>=NO)"
270 PRINT:INPUT R$:IF LEFT$(R$,1)<>"S" THEN 310
280 IF SW THEN GOSUB 3500:REM ILUSTRA JUGADA ANTERIOR
290 GOSUB 4000:REM ILUSTRA SITUACION
300 REM
310 SW=1:GOSUB 1000:REM PUSH ESTADO
320 REM EJECUCION JUGADA
330 IF DIR=0 THEN GOSUB 2000
340 REM JUGADA EN ORILLA DERECHA
350 IF DIR=1 THEN GOSUB 2500
360 REM JUGADA EN ORILLA IZQUIERDA
370 IF FIN THEN 610:REM RESUMEN Y FIN
375 IF FM THEN GOTO 580
380 GOSUB 3000:REM VALORACION COMILONA
390 IF NOT C THEN DIR=NOT DIR:GOTO 270
400 REM C=COMILONA SI NO -C INVIERTE RUTA
410 PRINT "MUMBLE-MUMBLE...ESTO NO SE
    SE PUEDE HACER ":PRINT
420 REM SI C=1 EXPLICA LA RETIRADA
430 IF DIR THEN 510 :REM EXPLICACION
    PARA ORILLA IZQUIERDA
440 REM EXPLICACION PARA ORILLA DECHA
450 PRINT "SI CARGO EN BARCA ";AN$(PAS)
460 AN$(M);" SE COME HA ";AN$(V)
470 PRINT "EN LA ORILLA IZQUIERDA..."
480 GOTO 560
490 REM
500 REM DIR=1,O SEA.ORILLA IZQ.
510 PRINT "SI DEJO EN LA ORILLA IZQUIERDA"
520 IF PAS=0 THEN 540
530 PRINT "CON ";AN$(PAS);" A BORDO"
540 PRINT AN$(M);" SE COME A ";AN$(V)
550 PRINT "EN LA ORILLA IZQUIERDA"
560 PRINT "POR LO TANTO CAMBIO JUGADA":
    PRINT:PRINT
570 FOR I=1 TO 1000:NEXT I:REM PAUSA
```

Figura 6.—Continuación listado del programa.

```
580 IF FM THEN GOSUB 1200:REM POP
585 IF IS=0 THEN PRINT "FINAL DE JUEGO":END
590 GOTO 330:REM NUEVA JUGADA
600 REM RESUMEN Y CONCLUSION
610 PRINT:PRINT:PRINT "TRAYECTO ULTIMADO
    (DAR <RET> PARA EL RESUMEN)":GET R$
620 HOME:HTAB 14:PRINT "RESUMIENDO:":PRINT
630 PRINT "JUGADA";:HTAB 11:PRINT "SI";:HTAB 19
640 PRINT "DE LA";:HTAB 29:PRINT "A LA"
650 PRINT "NUMERO TRANSBORDO";
660 HTAB 19:PRINT "ORILLA";:HTAB 29:PRINT "ORILLA"
670 PRINT:SPEED=100
680 FOR I=2 TO IS:REM SE CEPILLA LA PILA
690 DIR=VAL (MID$(STK$(I),9,1))
700 PASS=VAL (MID$(STK$(I),10,1))
710 HTAB 3:PRINT I-1;:HTAB 9:PRINT AN$(PAS);
720 HTAB 19:IF DIR THEN PRINT "DERECHA  IZQUIERDA":
    GOTO 740
730 PRINT "IZQUIERDA    DERECHA"
740 NEXT I
750 PRINT:PRINT:PRINT:FOR I=1 TO 20000:NEXT I
760 PRINT "....Y VIVIERON FELICES Y CONTENTOS"
770 PRINT "EN EFECTO,NO SIEMPRE ES FACIL "
780 INVERSE:PRINT:HTAB 8:PRINT "SALVAR CABRA Y
    COLIFLOR":NORMAL
790 HTAB 10:PRINT "******************":SPEED=255
800 PRINT "¿OTRA VUELTA?":INPUT R$
810 IF LEFT$(R$,1)<>"S" THEN END
820 FIN=0:GOSUB 1200:REM POP
830 GOTO 330
999 REM PUSH
1000 IS=IS+1
1010 REM DATOS DE ESTADO EN CADENA
1020 STK$(IS)=""
1030 FOR K=0 TO 3
1040 STK$(IS)=STK$(IS)+STR$(OD(K))
1050 NEXT K
1060 FOR K=0 TO 3
```


Figura 6.—Continuación listado del programa.


```
1070 STK$(IS)=STK$(IS)+STR$(DI(K))
1080 NEXT K
1090 STK$(IS)=STK$(IS)+STR$(DIR)
1100 STK$(IS)=STK$(IS)+STR$(PAS)
1110 STK$(IS)=STK$(IS)+STR$(PD)
1120 STK$(IS)=STK$(IS)+STR$(PI)
1130 P=0
1140 RETURN
1150 REM
1200 REM POP
1210 FOR K= 1 TO 4
1220 OD(K-1)=VAL(MID$(STK$(IS),K,1))
1230 NEXT K
1240 FOR K= 5 TO 8
1250 DI(K-5)=VAL(MID$(STK$(IS),K,1))
1260 NEXT K
1270 DIR=VAL(MID$(STK$(IS),9,1))
1280 PAS=VAL(MID$(STK$(IS),10,1))
1290 PD=VAL(MID$(STK$(IS),11,1))
1300 PI=VAL(MID$(STK$(IS),12,1))
1310 DIR=NOT DIR:REM VUELVE SOBRE TUS PASOS
1340 IS=IS-1:REM DESINFLA PILA
1350 P=1
1360 RETURN
1990 REM JUGADA EN ORILLA DERECHA
2000 IF NOT PAS THEN 2030
2010 OD(PAS)=1:REM DECARGA PASAJERO
2020 OD(0)=OD(0)+1
2030 IF P=0 THEN PD=0
2040 PD=PD+1:REM NUEVO PASAJERO
2050 FM=PD=4:IF IF FM THEN RETURN
2060 IF PD=PI OR OD(PD)=0 THEN 2040
2070 PAS=PD:OD(PD)=0:OD(0)=OD(0)-1
2080 RETURN
2090 REM
2490 REM JUGADA EN ORILLA IZQUIERDA
2500 IF NOT PAS THEN 2530
2510 OI(PAS)=1:OI(0)=OI(0)+1:REM DESCARGO PASAJERO
```

Figura 6.—Continuación listado del programa.

```
2520 IF OI(0)=3 THEN FIN=1:RETURN
2530 IF P=0 THEN PI=0:GOTO 2590
2540 REM PRUEBA A VOLVER VACIO
2550 PI=PI+1:REM P=1,VUELVE A LLEVAR A ALGUIEN
2560 FM=PI=4:IF FM THEN RETURN
2570 IF PI=PD OR OI(PI)=0 THEN 2550
2580 OI(PI)=0:OI(0)=OI(0)-1
2590 PAS=PI
2600 RETURN
3010 P=0:REM HIPOTESIS OPTIMISTA
3020 IF DIR THEN 3100
3030 REM ANALISIS ORILLA DERECHA
3040 IF OD(0)<>2 THEN RETURN
3050 IF OD(1) AND OD(3) THEN RETURN
3060 P=1:REM COMILONA SEGURA
3070 IF OD(1) AND OD(2) THEN J=1:V=2
3080 IF OD(2) AND OD(3) THEN J=2:V=3
3090 RETURN
3100 REM ANALISIS ORILLA IZQUIERDA
3110 IF OI(0)<>2 THEN RETURN
3120 IF OI(1) AND OI(3) THEN RETURN
3130 P=1:REM COMILONA SUPER SEGURA
3140 IF OI(1) AND OI(2) THEN J=1:V=2
3150 IF OI(2) AND OI(3) THEN J=2:V=3
3160 RETURN
3170 REM
3180 REM
3990 REM ILUSTRA LA SITUACION COMPLETA
4000 PRINT "AHORA LAS COSAS ESTAN ASI:"
4010 PRINT "ORILLA DERECHA:";:IF OD(0)=0
     THEN PRINT AN$(0):GOTO 4070
4020 FOR I=1 TO 3
4030 IF OD(I) THEN PRINT AN$(I);" ";
4040 NEXT I
4050 PRINT:IF DIR OR NOT PAS THEN 4070
4060 PRINT "MAS,AHORA, ";AN$(PAS):PRINT
4070 PRINT "ORILLA IZQUIERDA:";:IF OI(0)=0
     THEN PRINT AN$(0):GOTO 4110
```


Figura 6.—Continuación listado del programa.


```
4080 FOR I=1 TO 3
4090 IF OI(I) THEN PRINT AN$(I);" ";
4100 NEXT I
4110 PRINT:IF DIR OR NOT PAS THEN 4130
4120 PRINT "MAS,AHORA, ";AN$(PAS):PRINT
4130 RETURN
4140 REM
4150 REM %%%%%%%%%%%%%%%%%%%%%%%%%%%
4160 REM %%%  FIN DEL PROGRAMA  %%%
4170 REM %%%%%%%%%%%%%%%%%%%%%%%%%%%
```


Figura 6.—Continuación listado del programa.

Después de las líneas de inicialización 70-230, que damos por obvias, especialmente si se consulta la figura 4, aparece el POLLING DE USUARIO (líneas 240-300), también fácil de entender con la ayuda del pequeño flujo homónimo de la figura 5. En ambos se aprecia el papel del indicador del primer giro SW: sirve para saltarse al principio la ilustración de una (aún inexistente) jugada precedente. Es también un ejercicio útil y sencillo seguir los mensajes de las subrutinas 3500 y 4000 y los de justificación de la "retirada", desdobladas para las dos orillas, en las líneas 430 a 570. El planteamiento estructural de los datos resulta particularmente claro. Por ejemplo:

```
410 PRINT "MUMBLE,MUMBLE,ESTO NO SE PUEDE HACER"
420 REM SI P=1 EXPLICA RETIRADA
430 IF DIR THEN 510:REM EXPLICACION PARA ORILLA IZQ.
440 REM EXPLICACION PARA ORILLA DER.
450 PRINT "SI CARGO EN LA BARCA ";AN$(PAS)
460 PRINT AN$(M);" SE COME A LA ";AN$(V)
470 PRINT "EN LA ORILLA DERECHA"
480 GOTO 560
.........
560 PRINT "POR LO TANTO,CAMBIO DE JUGADA":PRINT:PRINT
```

Los parámetros P (0 ó 1), C y V se generan en la subrutina (desdoblada, como de costumbre) VALORACION COMILONA (GOSUB 3010). En ella el RETURN es inmediato si no se dejan en la orilla dos sujetos o bien si se trata del caso inocuo lobo+coliflor; en otra situación en C y V se pondrán los números correspondientes al individuo "comedor" y al "víctima". Ejemplo:

```
3070 IF OD(1) AND OD(2) THEN C=1:V=2
```

Es decir, la presencia simultánea del lobo y de la cabra asigna a los animales 1 y 2 los roles C y V respectivamente.

El meollo del problema

Las parejas de subrutinas 2000, 2500 (de gestión de la jugada en las dos orillas) y 1200 (los conocidos PUSH y POP) son el verdadero meollo. Las primeras dos se parecen en muchas cosas, por ejemplo, que ambas inicialmente se prevé que, en la condición NOT PAS, se evite un desembarco inútil pero que se simule incrementando en una unidad OD(0) u OI(0), que contarán los que permanecen en tierra, y situando un "1" en OD(PAS) u OI(PAS). Después se intentará con un nuevo pasajero mediante los índices PD o PI. Por ejemplo, en la orilla derecha:

```
2040 PD=PD+1:REM NUEVO PASAJERO
2050 FJ=(PD=4):IF FJ THEN RETURN
2060 IF PD=PI OR OD(PD)=0 THEN 2040
2070 PAS=PD:OD(PD)=0:OD(0)=OD(0)-1
2080 RETURN
```

El pequeño bucle de las líneas 2040-2060 acaba por fuerza, bien con la condición FJ="on" (provocada, a su vez, por la llegada a PD=4, o sea, por agotar todas las jugadas teóricamente posibles, desde "nadie" hasta "coliflor"; es lo mismo FM=Final de movimiento, que FJ=Final de jugada) bien con el descubrimiento de un pasajero PD comible, que (línea 2070) será embarcado en PAS, anulando OD(PD) y decrementando el cuenta-pasajeros OD(0). En la línea 2060 aparece toda la casuística del "pasajero imposible", es decir, por qué se excluye el volver a transportar a quien se acaba de desembarcar (dése cuenta que después del primer minibucle es más limpio usar PI en lugar de PAS) o, simplemente, por qué el pasajero en cuestión NO está. En lo que se refiere a la orilla izquierda, la imperfecta especularidad (o simetría) del código se hace evidente de esta forma:

```
2500 IF NOT PAS THEN 2530
...........
2530 IF P=0 THEN PI=0:GOTO 2590
2540 REM INTENTA DE VOLVER DE VACIO
...........
2590 PAS=PI:REM EMBARQUE
2600 RETURN
```


Llegados a este punto dejaremos que el paciente lector encuentre en el programa principal el camino por el que se llega a la ya citada valoración "comilona" que, por ejemplo, podría suponer la renuncia al intento de volver de vacío porque la cabra se comería la coliflor.

Ha llegado el momento de ver el PUSH y el POP de la matriz STK$ que reemplaza la pila (subrutinas 1000 y 1200). El mecanismo elegido, no demasiado difícil de rastrear en el listado, consiste en introducir con el PUSH y recuperar mediante el POP (con la función MID$) una cadena de estado STK$. Esta será resultado de la concatenación sucesiva de: OD(0), ..., OD(3), OI(0) ,..OI(3), DIR, PAS, PD y PI. El índice IS es incrementado al principio de la subrutina de PUSH, mientras que en la de POP la instrucción IS=IS-1 está puesta al final. En otros términos: con el PUSH, antes que nada, se crea un nuevo espacio en la matriz STK$; con el POP, por el contrario, se extraen los datos al principio. Observemos, además, que la condición IS=0 corresponde a la pila vacía.

En cuanto al por qué de los ajustes (semi-empíricos) DIR=NOT DIR y P=1 de las líneas 1310 y 1350 se trata de un pequeño rompecabezas que dejamos resolver al lector, un poco por pereza y un poco por sadismo. Los más adelantados sabrán que una pila implementada mediante una matriz (o vector) se debe considerar como una pseudopila (opera con un índice en vez de con un puntero). La elección era obligada en BASIC, pues este lenguaje no posee semejante estructura. De todas formas, en nuestro caso, su relativa "impureza" nos ha permitido una pequeña ventaja. En efecto, una memoria LIFO no podría recorrerse directamente, a menos que no se descargara antes... en una matriz. Al llegar a la condición FIN "encendida" en la línea 2520 (obviamente equivalente a OI(0)=3) se inserta fácilmente la fase de resumen (línea 600). La visualización de datos, como se puede deducir y comprobar es del tipo siguiente:

RESUMIENDO:

JUGADA NUMERO	SE TRANSPORTA A	DE LA ORILLA	A LA ORILLA
1	LA CABRA	DERECHA	IZQUIERDA
2	NADIE	IZQUIERDA	DERECHA
3	EL LOBO	DERECHA	IZQUIERDA
....	....	....	....
7	LA CABRA	DERECHA	IZQUIERDA

Es muy sencillo darse cuenta de que barriendo la matriz STK$ desde 2 hasta IS, se consigue reconstruir las siete jugadas —mediante MID$ (STK$(I), 10,1)— que están guardadas e, incluso, los di-

ferentes DIR (para escribir alternativamente "DERECHA" e "IZQUIERDA".

Incluso con una reconstrucción "en frío" (aunque siempre será mejor experimentando con el programa, o sus variantes, en un ordenador) resulta sencillo seguir los mensajes y reflexiones "en voz alta" que hemos examinado.

En la figura 7, según lo que ya adelantamos, están representados los movimientos de la pila; por simplicidad se ha representado, para cada nivel; sólo el dato PAS (expresado por su nombre: CABRA, COLIFLOR, NADIE, etc. para mayor evidencia). Se notará que el primer paso comporta solamente operaciones de push.

Una pequeña (¿o gran?) paradoja

Nos queda una curiosidad: ¿consigue nuestro programa barrer enteramente el árbol y hallar el camino de las siete jugadas? ¿Cómo?

Bien, la respuesta a la primera pregunta es negativa. Quienes escribimos tenemos que admitir nuestra sorpresa cuando, después de las no pocas penas de la investigación y puesta a punto, nos encontramos, DESPUES DE LA PRIMERA BUSQUEDA, con éxito (como ya se ha visto) frente a la desconcertante visualización de unos datos que definían como vía alternativa una serie de trece jugadas. Estas, a la tercera búsqueda, se convertían en diecinueve e iban aumentando a cada posterior respuesta positiva a la pregunta "¿OTRA VUELTA?"

Tenemos que confesar que, hasta entonces, no habíamos trazado ningún árbol del aparentemente banal juego. Además de rápida, esa elección nos pareció, de alguna forma "honrada": en efecto, queríamos un programa que nos "ayudara" a deshacernos de un árbol como si nos fuese desconocido a ambos. Si, con un cierto sentido del "después", nos remitimos a lo expuesto en la figura 1 y con la ayuda de la figura 7 (que evidencia la secuencia de push y pop de éste y del caso anterior) la explicación resulta evidente rápidamente: el ordenador, después de haber encontrado la estrategia de siete jugadas "cb, n, l, cb, co, n, cb" en la segunda prueba habrá llegado con dos pop al penúltimo nudo "S" y, como alternativa a la elección anterior "n", probará ahora con "l", después de lo cual seguirá sin dificultad hasta el nudo "+" situado al final de la figura. Las jugadas se sucederán ahora en el siguiente orden: cb, n, l, cb, co, l (en vez de n), cb, co, l, cb, co, n, cb, para un total de trece viajes.

El problema, por decirlo así, es doble:

- con el programa visto se encontrarán soluciones cada vez más largas;

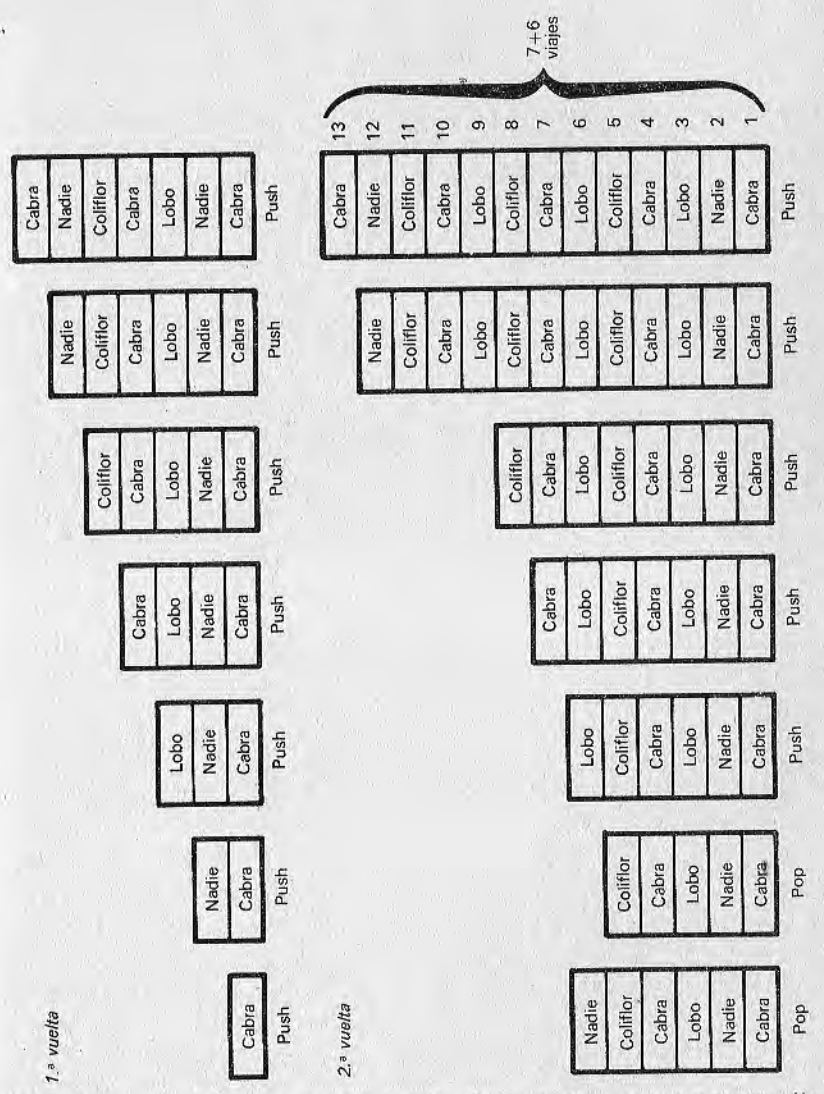


Figura 7.—Movimientos en la pila. Por sencillez se representa sólo el dato asociado al pasajero cargado en cada caso. En el primer viaje, de 7 pasos, se realizan sólo operaciones de PUSH. En el segundo giro (d) sólo se señalan, para mayor claridad, las 5 primeras y las 2 últimas condiciones. Después de dos POP nos encontramos un nudo donde, en lugar de la jugada hecha en la primera vuelta (NINGUNO), se elige la segunda alternativa (LOBO). El resultado son 6 viajes de más con respecto al primer caso. El árbol de este simple juego se revela infinito y no es por tanto factible representarlo de una forma exhaustiva.

- el mecanismo, sin embargo, es correcto, pues el aparentemente escuálido árbol, en realidad es infinito.

Pero entonces, sin las oportunas modificaciones ¿es imposible encontrar otra estrategia en siete jugadas? Bueno, ante todo, existen dos soluciones (un poco chapuceras, eso sí). La primera consiste en modificar el orden de los pasajeros (coliflor, lobo, cabra, por ejemplo) y, coherentemente, algunas de las líneas referentes a las jugadas y a la valoración de la comilona (si no, el programa, imperturbable, nos dirá: "mumble, mumble, la cabra come al lobo" u otras frases parecidas). Este artificio permite, que se nos haga evidente la otra solución: consiste en añadir otro pop a la línea 820 (probarlo para creerlo):

```
820 FINAL=0:GOSUB 1200:GOSUB 1200:REM DOBLE POP!
```

A partir del árbol de la figura 1, puede darse cuenta que este brusco truco hace que se salte el nudo S ya comentado y, dado que los otros dos de arriba están "saturados", se hará con ellos un POP automático, con lo cual se podrá llegar al nudo D en el que el ordenador escogerá, en un segundo momento, coliflor en lugar de lobo. A la tercera vuelta, incluso, podrá subir a la raíz y declarar el final del juego por falta de otras vías optimizadas.

Estas soluciones, por desgracia, no consiguen satisfacer nuestros escrúpulos lógicos.

Existe una solución correcta; consiste en descartar vías cuya longitud sea mayor que la mínima encontrada anteriormente. Como de costumbre, lo dejaremos como ejercicio para los más adelantados.

Terminamos ahora con unas consideraciones sobre las grandes dificultades que se encuentran en el nuevo y fascinante mundo de la llamada Inteligencia Artificial.

Incluso de este pobre ejemplo puede nacer la atroz duda de que sea ciertamente terrible, pero "orgánicamente" imposible, intentar remediar las contradicciones de fondo entre los mecanismos generales de deducción y las exigencias de problemas específicos. Hacemos notar a los más perceptivos que, incluso en las pequeñas elecciones que hemos hecho anteriormente (como la de intentar volver de vacío de la orilla izquierda), no se consigue entender como se podrían expresar, o "salir por ellas mismas" con la magia de lenguajes tipo LISP y Prolog. En suma, un Resolutor Generalizado que sea al menos un poco eficiente y que evite (sin necesidad de intervenciones penosas del programador) trampas como la de los recorridos infinitos, está hoy en día aún en los límites de la Utopia.

Este tema, empero, se sale de nuestros muy reducidos límites. Señalando que los investigadores en el campo de la Inteligencia Artificial han sabido escoger artificios y mecanismos que utilizan para evitar o, por lo menos, limitar peligros semejantes, debemos en todos los casos admitir que los desafíos en estas categorías de problemas, son fascinantes y estimulan ideas posteriores. ¡Incluso para los que se tienen que apañar con el BASIC!

CAPITULO V

ALGUNOS CONSEJOS SOBRE CUESTIONES PRACTICAS

El complicado tema de programar correctamente ha sido desmenuzado hasta ahora siguiendo una serie de principios. Estamos seguros de que alguien los clasificará como "teóricos", pero no quisiéramos iniciar una polémica entre los partidarios de la Teoría y los de la Práctica; por lo tanto, nos limitamos a observar que, para ser teóricos, los ejemplos que hemos ofrecido han sido ricos, concretos y abundantes. Se referían a temas y problemas de planteamiento con los cuales no pocos de los habilísimos seguidores de la informática que encontramos hoy en día, se afanan penosamente sin resultado y sin saber, muchas veces, ni siquiera por donde empezar.

Por eso insistimos en que obsesionarse con los algoritmos es un ejercicio penoso y frustrante muchas veces, aunque puede dar satisfacciones mucho mayores que el ejercicio de la PEEK-POKE-manía. Con esta etiqueta nos permitimos ironizar (perdón) sobre ciertos fanáticos del ordenador personal que abusan de las útiles instrucciones PEEK y POKE del BASIC para acceder al lenguaje máquina y a las subrutinas del sistema operativo. Habrán notado que nosotros las hemos evitado con el mayor cuidado, con el fin de obtener la máxima "transportabilidad", es decir, la mayor equivalencia posible entre programas para un ordenador y otro. Debemos notar que en cuanto tocamos el terreno pragmático nos damos de cara con las contradicciones más atroces: en efecto, la transportabilidad es una exigencia práctica, pero también lo es aprovechar un sistema al máximo de sus potencialidades y, sin embargo, ambas son opuestas...

Es necesario reconocer que, desde un punto de vista opera-

tivo, se han dejado al margen bastantes cosas pertenecientes a la realidad de las máquinas y de los problemas de todos los días. Pero también se han dicho varias cosas, en modo indirecto o implícito, no sólo en este libro sino también en el de la programación estructurada (BBI n.º 8); por no hablar de los volúmenes sobre el lenguaje BASIC (BBI n.º 5, 6 y 7) a los cuales les remitimos para minucias tales como el uso de GET, como alternativa a la INPUT (habrán notado que, por pereza, la instrucción GET no la hemos utilizado nunca aquí...). Pero hay otros problemas prácticos que se deberían afrontar... en línea teórica (¡NO es una finura, sino una contradicción real en los terminos!). Aludimos, por lo menos, a dos grandes categorías:

- cómo organizar un adecuado interface con el usuario, o sea, un "menú" apropiado para definir las opciones a quien vaya a utilizarlo, que resulte claro, sencillo y, como dicen los americanos, "fool proof" (a prueba de tontos);
- cómo hacer más rápido y eficiente un programa, especialmente en relación al microsistema que se posee y al lenguaje utilizado.

Ambos problemas se agravan en el caso de programas muy largos, que buscan la seriedad multiplicando y complicando los dilemas. Estos nos remiten a una cuestión delicada que, a grandes rasgos, se expresa como sigue:

¿Hasta qué punto es conveniente la adopción de "trucos" que, con un uso astuto de la máquina y de sus peculiaridades específicas la aprovechen de la mejor manera?

Una contraindicación —la de la transportabilidad— la hemos visto ya. Se pueden dar sugerencias y consejos útiles con la finalidad de un llegar a un compromiso razonable y válido en casi todos los ambientes y sistemas.

Por otra parte, la mayor dificultad reside precisamente en proporcionar ejemplos concretos.

Antes de pasar a desmenuzar la segunda categoría de problemas, vamos a permitirnos dar un par de apuntes referentes a la primera categoría (interface con el usuario), apuntes que cada uno deberá acoplar a su particular experiencia personal a través de la práctica del "prueba y vuelve a probar", que ningún tratado o manual pueden reemplazar.

En el caso de un menú de elecciones múltiples puede resultar más claro para el interlocutor del programa que, en lugar de encontrarse delante de los habituales numeritos:

1. Editar	2. Imprimir
3. Transferir	4. Cargar
5. Terminar	6. ....

encuentre, en cambio, como propuesta las letras iniciales de las diferentes opciones, o sea, algo así como:

E = Editar	I = Imprimir
T = Transferir (al disco)	C = Cargar (desde disco)
A = Abandonar	X = ...

(dénse cuenta de los ajustes de sinónimos necesarios para diferenciar las iniciales de las opciones).

La ventaja de las órdenes nemónicas es que permiten que el usuario recuerde fácilmente el código que corresponde a cada opción sin necesidad de recurrir al mensaje del menú. Se implementan muy bien en lenguajes que, como el Pascal, ofrecen la estructura CASE OF. ¿Y en el BASIC? Aquí, la estructura casi equivalente ON...GOSUB es un poco menos potente, dado que trabaja cómodamente con números, pero no acepta letras. Se puede remediar mediante un pequeño vector que contenga las diferentes inciales. Supongamos que lo llamamos INIC$ y que R$ contiene la inicial elegida por el usuario; la respuesta del ordenador podría funcionar mediante las líneas BASIC que siguen:

```
300 ENCU=FALSO
310 FOR I=1 TO NUMOPC
320 IF R$=INIC$(I) THEN ENCU=VERDADERO:K=I:I=NUMOPC
330 NEXT I
340 IF NOT ENCU THEN GOTO X
350 ON K GOSUB R,S,T,......
```

Es obvio que, r, s, t, etc., serán los números de línea de las diferentes subrutinas de respuesta, mientras que x será el de la línea que contiene el INPUT. Así hemos satisfecho (naturalmente es un ejemplo muy sencillo) el principio de reducir las posibilidades de error por parte del usuario. En programas más largos y/o importantes será oportuno prever, también, barreras defensivas contra las más extrañas faltas de atención (apretar teclas críticas, hacer maniobras que hagan volver al sistema operativo, etc.), posibles siempre en el transcurso de sesiones largas y fatigosas. Afortunadamente, en todos los dialectos BASIC existe la cómoda instrucción ON ERROR GOSUB (o GOTO) que pone remedio a muchos de tales desgracias (para aprender sus diferentes modalidades de uso les remitimos a los manuales de los ordenadores y a los volúmenes dedicados a este lenguaje, donde se encuentran descritos adecuadamente).

A propósito de teclas y menús debemos hacerles otra recomendación referente al caso de programas en los que se encuen-

tran varios menús y submenús, o sea, a aquellos organizados "por árbol": intente adoptar siempre la misma semántica en todas las circunstancias, evitando que en un menú Imprimir se exprese con "I" mientras que en otro se exprese con la letra "P" (de Print) o con el número "2". Otro consejo: utilice las teclas combinadas con la de Control y, para las condiciones de "quit" (abandono de programas), usen sistemáticamente la tecla de Escape.

Especialmente en programas de cálculo científico y técnico será una buena norma introducir controles de congruencia de variables y orden de magnitudes, pero tenga mucho cuidado con las llamadas "modalidades de restablecimiento". Veamos un ejemplo tonto, que es más caricaturesco:

```
1500 IF DATO<INF OR DATO>SUB THEN END
```

Aquí, si el desafortunado usuario se equivoca introduciendo un DATO que se encuentra fuera de los límites definidos por INF y SUB se le penaliza gravemente con la paralización del programa y, lo que es más grave, no sólo con la pérdida de los datos generales por el programa, sino también con la de todos los introducidos por él hasta el momento.

¿Se trata de sadismo? Quizá, pero nos estamos refiriendo a un caso real (naturalmente la condición de paralización surgía de una forma más indirecta y fastidiosa): un señor compró un lote de programas de cálculos de ingeniería y, dándose cuenta de que había algo raro, rehusó efectuar el pago hasta que el problema no se hubiese solucionado; acabaron en Magistratura. No sabemos como se habrá resuelto, pero si estuviéramos en el lugar del juez hubiéramos dado toda la razón al comprador.

Acabamos recomendándoles que no tomen demasiado en cuenta los oropeles escenográficos. Aunque a veces no está de más un mínimo de videografía y algún efecto sonoro agradable, sobre todo para llamar la atención en el momento adecuado, a lo que se debe de prestar la máxima atención es a los posibles defectos. Sólo la experiencia enseña a corregirlos y, posiblemente, a prevenirlos.

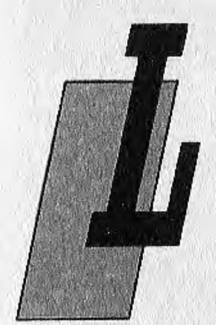
En los próximos dos capítulos trataremos bastante ampliamente los problemas de velocidad y eficiencia que plantea el BASIC (el lenguaje-caracol). Como resulta inevitable al tener que afrontar un tema muy concreto, debemos hacer referencia a un ordenador personal en carne y hueso (perdón, en plástico y silicio). La elección, por sus prestaciones, popularidad y difusión, ha recaido en el ordenador personal Commodore 64, llamado también C64. A grandes rasgos el tratamiento es válido para la gran mayoría de los ordenadores del mercado; por ejemplo, los problemas de la "información inservible" en las cadenas se pueden encontrar,

prácticamente sin cambios, en cualquier ordenador. Pasando las referencias al mapa de memoria de su propio microordenador será fácil para todos reciclar todos los conceptos (generalmente basta con adaptar las direcciones de los PEEK y POKE a aquellas en vigor en nuestro sistema).

CAPITULO VI

EL BASIC, UN LENGUAJE-CARACOL

Interpretación: una ventaja que sale cara



La enorme popularidad del lenguaje BASIC en el mundo de los pequeños sistemas (ordenadores domésticos y personales) está ligada sobre todo al hecho de ser, casi siempre, un lenguaje interpretado. Este adjetivo, como bien saben ustedes, quiere decir que cada instrucción —IF, GOTO, sentencias aritméticas, etc.— es traducida directamente a lenguaje máquina (de ahora en adelante escribiremos l. m.) en el momento de la ejecución ("run time") cada vez que esta se lleva a cabo, sin necesidad de compilaciones u otros procesos preliminares.

Aparentemente el ordenador se comporta como una máquina BASIC: comprende y ejecuta de manera inmediata órdenes en lenguaje evolucionado, pero en realidad, cada vez que lo hace realiza antes la traducción, aunque sin dejar rastro al final. Desde el punto de vista de la interacción hombre-máquina se trata, sin duda, de una ventaja notable, porque el sistema se puede manejar directamente en un código simbólico (de alto nivel, cercano al lenguaje humano) y no surgen los inconvenientes (y la pérdida de tiempo) que trae consigo la intervención de un programa compilador y el hecho de tener dos versiones de un mismo programa (desde un punto de vista lógico-funcional): el programa "fuente" (source program, el que podemos editar y listar), y el programa "objeto" (object program, el que está en l.m. traducido por el compilador al código máquina del microprocesador que constituye la CPU del sistema).

Con el BASIC interpretado se ejecutan más fácilmente, entre

otras cosas: controles, añadidos, modificaciones (en cualquier momento) y la búsqueda y eliminación de errores resulta más sencilla y rápida.

Sin embargo esta ventaja se ve fuertemente penalizada, porque se paga en términos de disminución de la velocidad de ejecución. Puede suceder, en efecto, que el tiempo de interpretación (que, repetimos, se invierte todas las veces que se ejecuta cada instrucción y no de una vez por todas como sucede con los lenguajes compilados) resulte mucho mayor que el correspondiente a su ejecución efectiva. El inconveniente existe, incluso, en las instrucciones más sencillas.

Pero la popularidad del BASIC también reside, en el tratamiento de las variables y en el manejo de la memoria interna: el programador no se tiene que preocupar para nada de su situación, ya que se halla enteramente bajo la supervisión del intérprete. Una vez más, se obtiene una prestación muy cómoda pero que supone ralentizar el proceso. En general, no estamos demasiado dispuestos a renunciar a estas ventajas, por lo menos dentro del ámbito de las auténticas aplicaciones de los ordenadores domésticos.

Afortunadamente, esta lentitud, derivada del proceso interpretativo, se puede obviar, por lo menos en parte, con una serie de pequeños recursos, cuya descrpción es nuestro próximo objetivo. Uno muy habitual es la inserción de pequeñas rutinas de l.m. dentro de los mismos programas BASIC; los códigos correspondientes a la posición de memoria inicial se introducen mediante POKE en una determinada situación de memoria, para ser recuperados en el momento oportuno con instrucciones específicas, como la USR y la SYS del C64, la CALL de otros microsistemas, etc. que provocan el salto a la subrutina en l.m. Esta técnica suspende momentáneamente la interpretación y pasa a la ejecución directa en l.m., permitiendo así aumentar la velocidad en ciertos pases críticos de un programa. No nos vamos a extender sobre ella por dos buenos motivos:

- está abundamentemente descrita en todos los manuales y libros;
- anulan la transportabilidad de un programa: entre máquinas que adoptan el mismo dialecto BASIC pero que se basan en CPU difeentes es imposible la transferibilidad de programas con SYS o CALL, mientras que con CPU iguales pero con ordenadores "diseñados" de forma diferente; se puede conseguir sólo con modificaciones radicales.

Nuestro objetivo es descubrir, sobre todo de forma experimental, todo lo que se puede hacer permaneciendo dentro del BASIC vulgar y (casi) estándar: las ventajas en términos de velocidad y, de alguna manera, de eficiencia serán pequeñas si las tomamos de una en una, pero cuando están en juego programas bastante largos y/o bucles con numerosas iteraciones, el tiempo que se ha ahorrado puede llegar a ser importante.

Tanto los inconvenientes como los remedios sugeridos, se ilustrarán cuando entremos en detalles sobre el mecanismo de la interpretación y sobre los criterios de memorización de las variables en la memoria RAM disponible. En cuanto a las pruebas que, por supuesto, les invitamos a realizar directamente, serán ejecutadas por nosotros en un sistema Commodore 64, pero todos podrán hacerlas, con fáciles modificaciones, en el ordenador que tengan en casa.

En relación con los resultados de las pruebas que hemos llevado a cabo, en líneas generales se pueden considerar válidos (naturalmente para igual frecuencia del reloj interno) para todas las máquinas que utilizan como CPU el microprocesador 6502 y derivados (el 6510 en el caso del C64). Quien haga las pruebas en el ordenador Commodore VIC 20 tendrá la agradable sorpresa de encontrarse con tiempos inferiores a los del más dotado C64. Esto no debe sorprender mucho si se piensa en el duro trabajo que debe realizar el chip de control de vídeo del C64 que, si por una parte permite visualizaciones más refinadas, por otra interfiere constantemente con el trabajo de la CPU principal 6510, haciendo más lentos los accesos a la memoria y los tiempos de ejecución de algunas instrucciones. En suma, para una utilización puramente de cálculo, el modesto VIC 20 gana en velocidad a su hermano mayor, mejor dotado.

Para hacer comparaciones entre la velocidad de distintos programas no hay nada mejor que usar el reloj interno del C64; esto mismo vale para otras máquinas que posean también reloj interno, en otros casos habrá que arreglarse con un cronómetro externo. En el Commodore 64 el valor del tiempo se puede encontrar utilizando dos variables especiales (reservadas): TI y TI$, la primera numérica y la segunda de tipo cadena (alfanumérica). Dado que TI$ tiene una resolución bastante escasa (un segundo), no se adapta a nuestros fines. En cambio TI puede considerarse suficientemente precisa, siendo capaz de discernir hasta 1/60 de segundo, o sea a 16,6 milésimas de segundo. La verdad es que los tiempos de ejecución de las instrucciones BASIC son inferiores, pero ésto no tiene una importancia excesiva, en cuanto que lo que pretendemos no es tanto el medir con exactitud la duración, sino comparar los resultados obtenidos con distintas técnicas de programación. Ante todo trataremos que el número de instrucciones ejecutables sea suficientemente elevado, lo que contribuirá a acentuar las diferencias, haciéndolas más apreciables.

Recordamos que la variable reservada "TI" no puede ser ini-

cializada desde el software. Pero no importa, será suficiente con copiar al principio su valor en una variable auxiliar (o depósito, como se quiera decir); al final, el valor del tiempo transcurrido lo obtendremos como diferencia entre el valor final de TI y el guardado inicialmente.

Memoria necesaria para el programa BASIC

Un aspecto importante que hay que tener en consideración si se quiere utilizar de la mejor forma posible un ordenador es, sin duda, la cantidad de memoria necesaria para que un programa pueda trabajar. Utilizando el BASIC interpretado se deberá tener en cuenta tanto el espacio de RAM ocupado por el código (el verdadero y propio programa) como el utilizado por las variables en juego. Aparentemente, un programa BASIC ocupa tanto espacio como número de caracteres componen sus líneas; en otros términos, pecisa tantos bytes como teclas de caracteres se hayan utilizado en la redacción del programa. En realidad las cosas discurren de forma un poco distinta.

Veamos cómo y por qué. Para mayor comodidad del lector, ilustramos en la figura 1 el mapa de memoria del C64, junto a la del VIC 20, con particular atención a la utilización que hace el BASIC de las primeras 256 posiciones de memoria (la llamada página cero). Para más detalles les remitimos al apéndice Q del manual de instrucciones del C64.

Nos daremos cuenta bastante pronto de que es fácil verificar cuánta memoria es utilizada y cómo. Una vez encendido el ordenador escriba:

```
PRINT PEEK(44)*256+PEEK(45)
```

La respuesta será 2049, es decir, la dirección en la cual la primera instrucción del programa BASIC es introducido. En otros ordenadores personales las localizaciones de la página cero que apuntan a este lugar de entrada, no serán 44 y 45, pero el proceso será el mismo; sólo necesitamos saber que son dos (44 y 45 aquí) los bytes necesarios para contener el valor de la dirección, en uno (44) se guarda la parte más significativa (256 vòces más) y en el otro (45) la menor.

Después, sin hacer nada más, podemos pedir la dirección de final del programa haciendo:

```
PRINT PEEK(46)*256+PEEK(47)
```

(A los bytes 46 y 47 se aplican los mismos comentarios realizados antes con 44 y 45).

POSICION		
DEC	HEX	
0	0000	Registro dirección datos del 6510
1	0001	Registro 8 bits de entrada/salida del 6510
2	0002	No usado
40-41	0028-0029	38÷42 Area usada para la multiplicación real
42	002A	
43 44	002B 002C	Puntero principio del texto BASIC
45 46	002D 002E	Puntero final programa y principio variables
47 48	002F 0030	Puntero principio vectores (matrices)
49 50	0031 0032	Puntero final vectores (+1)
51 52	0033 0034	Puntero principio variables cadena
53 54	0035 0036	Puntero final variables cadena
55 56	0037 0038	Puntero en el límite de la memoria para programas BASIC
136÷140	0088÷008C	Aquí las correspondientes localizaciones son 139÷143
197	00C5	Valor de la tecla pulsada actualmente
198	00C6	Número de caracteres en el buffer del teclado (cola)
631÷640	0277÷0280	Cola del buffer del teclado (F1F0)
650	028A	Indicador programable de repetición para las teclas (0 = cursor, 128 = todas)
653	028D	Indicador tecla SHIFT/CTRL/
828÷1019	033C÷03FB	Buffer de E/S de la cinta
1020÷1023	03FC÷03FF	Libres
1024÷2023	0400÷07E7	Area de memoria de pantalla (1024 bytes) matriz pantalla (25 líneas × 40 columnas)
2040÷2047	07FB÷07FF	Punteros a los datos de animación

POSICION DEC	HEX	
2048÷40959	0800÷9FFF	Area RAM normal de los programas BASIC, con punteros es posible "insertar" en el área (32768-40959) cartuchos ROM (8 k)
40960÷49151	A000÷BFFF	8 k ROM del intérprete BASIC (es posible usar RAM aquí para utilizarla con lenguaje máquina)
53248÷54271	D000÷D3FF	Controlador interface pantalla (VIC II) MOS 6566
54272÷55295	D400÷D7FF	Interface de sonido (SID) MOS 6581 3 osciladores independientes y programables con 4 clases de ondas, 9 octavas, etc.

Figura 1.—Mapa de memoria del Commodore 64. El texto hace referencia a las posiciones de la página cero. Para el ordenador personal VIC20 entas localizaciones son bastante parecidas (las diferencias se encuentran sobre todo en las direcciones más elevadas, por el distinto planteamiento del C64: gestión del sonido, de las animaciones, de los bancos de memoria RAM/ROM, etc.). Por lo que respecta a la organización de los tipos de datos (enteros, reales, de cadena) y su distribución en la memoria, el sistema ilustrado, referido al popular C64, es parecido, con pocos cambios, a los que usan la mayoría de los ordenadores personales.

Lo lógico sería esperar el mismo número de antes, ya que todavía no hemos introducido ninguna línea BASIC. En cambio obtenemos el valor 2051. No se asusten, no ocurre nada extraño; en efecto, dos bytes están comprometidos desde el principio por el BASIC para guardar el valor binario cero, indicador del final del programa (00 00 = final del programa).

Si ahora introducimos una simple línea podemos comprobar inmediatamente cuanta memoria ocupa:

```
1 REM ABCDEFGHIJK
```

La respuesta obtenida con el PRINT anterior es 2067, que corresponde a 18 bytes (2067-2049)ocupados desde el principio, y no a 15 como se podría suponer. Intentemos entonces visualizar el contenido de todas estas posiciones ocupadas:

```
FOR K=2049 TO 2067:PRINT CHR$(PEEK(K));:NEXT K
```

La respuesta es:

```
ABCDEFGHIJK
```


es decir, en apariencia sólo hay once caracteres. ¿Cómo es esto? Evidentemente, los otros 7 contienen valores binarios no visualizables en la pantalla. Haciendo la prueba con otros tipos de instrucciones la situación no cambia mucho. En particular, los nombres de las instrucciones (como REM) y los números de línea parecen haber desaparecido o ser sustituidos por extraños caracteres gráficos. La explicación de este aparente misterio reside en la forma en la que el intérprete memoriza el programa.

Cada una de las líneas del programa se representa en la memoria con la estructura que sigue (ver figura 2):

- 2 bytes que contienen la dirección de la siguiente instrucción;
- 2 bytes que contienen el número de línea;
- un byte para cada instrucción BASIC, representada según un código compacto que algunos llaman "token";
- N bytes tal y como hayan sido escritos desde el teclado;
- 2 bytes conteniendo ceros para señalar el final del programa, después de la última instrucción.

Se explican así los siete bytes aparentemente desaparecidos que hemos visto anteriormente. Se puede concluir, pues, que en principio la memoria que ocupa cualquier programa está dada por el número de caracteres tecleados (contabilizando, sin embar-

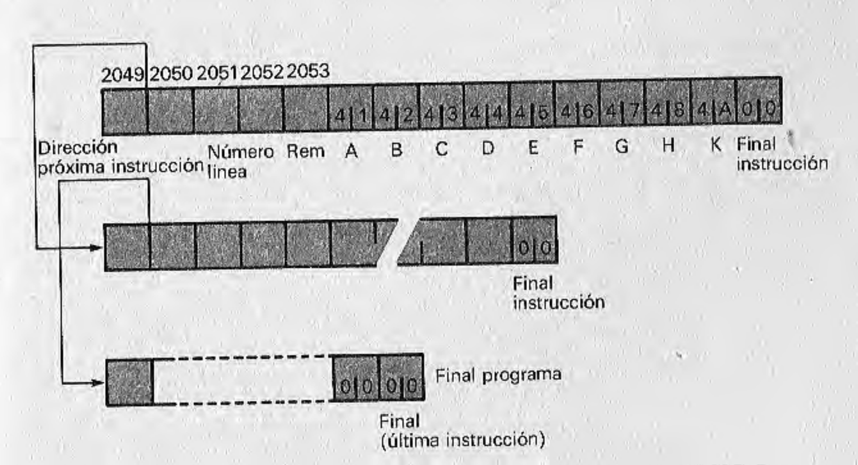


Figura 2.—Estructura y sucesión de las instrucciones BASIC. Para mayor comodidad se han suprimido los contenidos de las distintas posiciones; por ejemplo, el "token" ASCII del código REM, fácil de encontrar en el manual C64 (respecto al texto, faltan las letras "I" y "J").

go, un solo byte para los códigos operativos tipo REM, INPUT, DATA, +, –, *, etc,.), añadiendo 4 bytes por cada línea y contando los dos bytes finales de clausura.

Por consiguiente, si tiene problemas de espacio, puede intentar:

- compactar varias líneas en una sola, ganando así cada vez 3 bytes (uno es siempre necesario para los dos puntos de separación);
- eliminar todos los espacios de separación inútiles.

Debe de quedar bien claro que esta técnica, indispensable con programas muy largos destinados a ordenadores domésticos con no mucha memoria (por ejemplo el VIC 20 sin expansiones), no es compatible con exigencias de claridad. Hay además algunos dialectos que limitan su aplicación; por ejemplo, en el BASIC Microsoft los espacios de separación entre comandos y operadores son obligatorios (escribiendo PRINT A$ se obtiene de la máquina un "syntax error").

Espacio para las variables, cadenas y matrices

Una elección inteligente de la naturaleza y número de las variables a utilizar proporciona también resultados sorprendentes. Siempre con el ordenador recién encendido, examinemos el principio y el fin del área destinada a las variables, situada justo después del final del programa. Repitiendo:

```
PRINT PEEK(46)*256+PEEK(45)
```

la respuesta que se obtiene es 2051, dado que en este momento no hay presente ningún programa, y con:

```
PRINT PEEK(48)*256+PEEK(47)
```

la respuesta sigue siendo 2051, ya que no se ha definido aún ninguna variable.

Introduzcamos ahora:

```
AA=1
PRINT PEEK(48)*256+PEEK(47)
```

Esta vez se obtendrá el valor 2058. Si introducimos con X=12345678901 tendremos que el puntero de final de área de variables alcanza el valor 2065; la inmediata conclusión es que una

variable numérica, cualquiera que sea el valor que se le atribuya, ocupa siempre 7 bytes de RAM.

El motivo de esto reside en el hecho de que el BASIC, para este tipo de datos, adopta la notación de coma flotante (floating point), con nueve cifras significativas y exponente; por eso son necesarios 5 bytes para contener esas 9 cifras más signo y exponente, a los cuales se deberán añadir dos bytes para el nombre simbólico (en otros dialectos BASIC en los que el nombre de las variables puede estar formado por más de dos caracteres ASCII esta regla, por otra parte muy común en los ordenadores más usuales, no vale). Si, como en el último caso, éste está formado por una sola letra el intérprete añade, por su cuenta y riesgo, un espacio (blank). En consecuencia el único ahorro que se consigue usando nombres de un carácter reside en la longitud del programa.

Con la orden CLR se anulan todas las variables definidas, lo cual es fácilmente comprobable: el puntero de final de variables es llevado hasta 2051. Un consejo práctico: llevar hacia adelante artificialmente este puntero de final de variables (mediante una orden POKE) puede ser un método astuto y sencillo para recuperar los datos perdidos inadvertidamente.

El hecho de que la zona reservada a las variables empiece inmediatamente después del final del programa explica porqué, añadiendo instrucciones, los valores de las primeras variables son destruidos irremediablemente y cómo, con la técnica de "overlay" (segmentación de programas que no pueden estar contenidos enteramente en la RAM disponible) se pueden cargar sólo programas más cortos que el primero de la cadena.

La versión BASIC adoptada por el C64 no reserva un área específica para las variables enteras (de tipo "integer") y, por lo tanto, también estas ocupan siete bytes. Es una verdadera pena, puesto que su límite de amplitud (no superior al valor 65535, máximo alcanzable con 16 bits en la notación binaria adoptada) hubiera permitido la utilización de sólo cuatro bytes: dos para el nombre y dos para el valor. Probablemente, la buena disponibilidad de memoria ha inducido a los proyectistas a no cuidar este aspecto, presente en las otras máquinas de Commodore.

En cuanto a las variables alfanuméricas (cadenas) también éstas se memorizan como las anteriores, y así ocupan siete bytes en lugar de cinco, que serían suficientes: además del nombre (dos bytes) se memoriza la longitud (un byte) y la dirección de comienzo de la cadena (otros dos bytes). En efecto, la longitud de las cadenas es variable e imprevisible a priori, por lo que no sería posible memorizarlas directamente al lado del nombre, so pena de continuos desplazamientos de las variables subsiguientes en caso de cambiar el contenido de la cadena. El área en la que son

memorizados los valores de las cadenas comienza al final de la memoria disponible y se dirige hacia la ya ocupada. Este tipo de organización da lugar a una gestión eficiente y veloz, pero tampoco es inmune a todos los problemas, como veremos más adelante.

Ahora, alguien estará haciéndose esta inquietante pregunta: ¿cómo distingue el BASIC los diferentes tipos de variables, dado que estas se memorizan todas juntas, puestas simplemente en el orden temporal de definición? Elemental: el nombre de las variables numéricas se memoriza con el correspondiente código ASCII, mientras que en el caso de las variables enteras se añade 128 a la primera letra y, en el de las cadenas, se añade 128 a la segunda letra. Dado que el código ASCII sólo representa caracteres de código 0-127) (0-01111111 en binario) el añadir 128 a cualquier carácter ASCII equivale a poner "a 1" su bit más significativo (el situado más a la izquierda). Por ejemplo, el nombre PE (0101 0000 0100 0101 en binario) quedará memorizado como:

- en punto flotante:
  0101 0000 0100 0101
- entera:
  1101 0000 0100 0101
- cadena:
  0101 0000 1100 0101

El método es sencillo pero poco rápido, ya que la búsqueda de cada una es entorpecida por la presencia de todas las demás y no sólo por las del mismo tipo. En la figura 3 está ilustrada la organización de las variables sencillas de los tres tipos (coma flotante, entera y alfanumérica).

Inmediatamente después del final de las variables sencillas comienza la zona de los vectores (arrays). Para comprobar experimentalmente su colocación (e incluso antes de ver la figura 4, que ilustra la estructura de su organización) convendrá esta vez actuar de manera muy diferente a como lo hemos venido haciendo, valiéndonos de un sencillo programa, útil para cada tipo de vector. Este es:

```
 1 CLR:REM LIMPIEZA DE LA MEMORIA DE VARIABLES
10 IA=0:FA=0:REM PREDEFINICION VARIABLES DE TRABAJO
20 DIM RR(300):REM ARRAY DEFINIDO
30 IA=PEEK(48)*256+PEEK(47):REM PRINCIPIO AREA ARRAY
40 FA=PEEK(50)*256+PEEK(49):REM FINAL AREA ARRAY
50 PRINT FA,IA,FA-IA
```

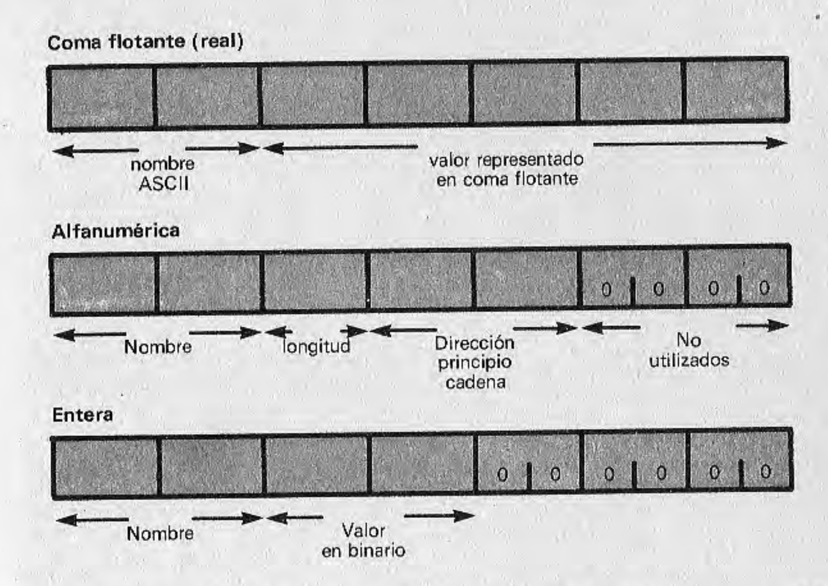



Figura 3.—Organización de las variables (al final del programa BASIC). En el caso de las cadenas, al nombre le siguen la longitud y un puntero que indica la dirección en la que la cadena tiene su principio. En el C64 los tres tipos ocupan 7 bytes (en otros dialectos de BASIC no siempre sucede esto).

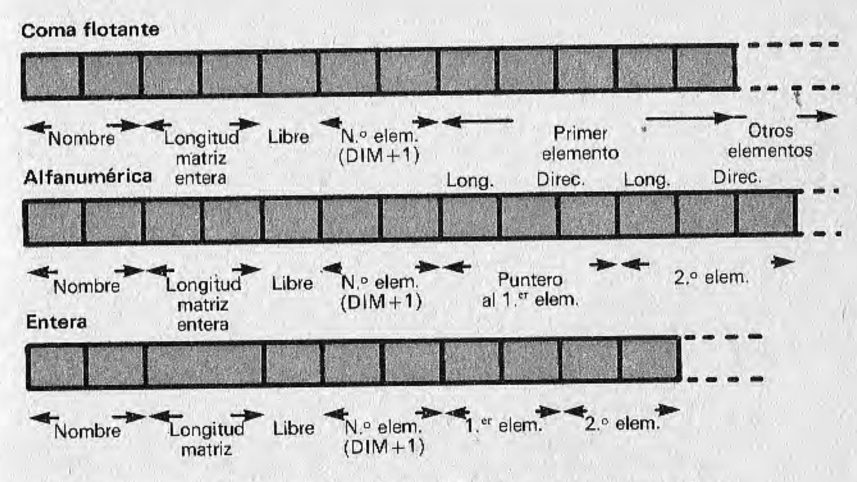


Figura 4.—Organización de los distintos tipos de matrices.

Una vez ejecutado, el programa escribe como tercer dato la memoria ocupada por la matriz, igual en este caso a 1512 bytes. Siendo los elementos en juego 301 (recuerden que también existe el elemento de índice cero) se deduce, sin demasiada dificultad, que cada uno de ellos emplea cinco bytes, como es necesario para una variable en coma flotante, más siete bytes de cabecera que contienen el nombre, el número de elementos y otras informaciones para uso del intérprete.

Cambiando ahora en la línea 20 "RR" por "RR%", que indica matrices de números enteros, el resultado será 609. Es decir: sin cambiar el número de elementos (301) cada uno de estos ocupa ahora sólo dos bytes, mientras quedan por añadir al total los siete bytes del encabezamiento común. En otros términos, esto significa que la utilización de variables de tipo entero supone un ahorro considerable de memoria.

Definiendo una matriz alfanumérica, poniendo en la línea 20 RR$(300), el resultado visualizado es 910, lo que significa que cada elemento de este tipo de matriz ocupa sólo tres bytes. El resultado, sobre todo teniendo en cuenta el hecho de que una cadena puede tener una longitud de hasta 256 caracteres, puede parecer paradójico, pero sólo a las personas poco atentas. Incluso estas no tendrán dificultad en comprender que, también en el caso de las matrices, las cadenas se encuentran en la parte alta de la memoria, mientras que en el espacio del vector cada elemento indica tan solo la longitud (un byte, por lo cual la longitud máxima es de 256 caracteres) y el puntero que indica comienzo de la cadena afectiva. De esta forma, con las cadenas, la ocupación de RAM crece proporcionalmente según a cada elemento —vacío en el momento del DIM inicial— se le va asociando una cadena real (no vacía).

Concluyendo, si puede trabajar con números enteros inferiores a 65533 conviene, sin duda, adoptar matrices de tipo entero, pudiendo así casi triplicar la memoria disponible para datos. En cambio —por lo menos en el caso del C64, como ya se ha visto— no hay ventaja alguna, en términos de espacio ganado, al usar variables enteras en lugar de las de coma flotante (en otras versiones BASIC, sin embargo, sí las hay).

Memoria y velocidad

Para comprobar experimentalmente cómo se las arregla el BASIC con las cadenas y, en particular, cómo influye su longitud en los tiempos de ejecución, les proponemos usar el programa siguiente con cronómetro incorporado. En él intercambiaremos tres

mil veces el contenido de dos cadenas entre sí, recurriendo a una cadena intermedia:

```
 1 CLR
10 A$="PRIMERA CADENA":B$="SEGUNDA CADENA":C$=" "
20 TD=TI
30 FOR K=0 TO 3000
40 C$=B$
50 B$=A$
60 A$=C$
70 NEXT K
80 PRINT TI-TD
```

Se obtiene un valor, correspondiente a múltiplos de 1/60 segundos (sesentavos), igual a 965. Probemos a reducir la longitud de las variables, haciendo, en la línea 10, A$="PRIMERA" y B$ "SEGUNDA".

Antes de proceder a la modificación y a la nueva medida, respondan: ¿cambiará la duración del bucle? La respuesta es negativa, contrariamente a lo que podrían imaginarse, pero la explicación de esta paradoja no es demasiado difícil. En efecto, bastará reflexionar nuevamente sobre la manera en que el intérprete BASIC sitúa las variables alfanuméricas: en el área reservada a tales variables no está colocado el contenido de la cadena, sino sólo su nombre, longitud y dirección a partir de la cual está escrita la cadena. Cambiar los datos alfanuméricos, por lo tanto, simplemente significa cambiar tales longitudes y direcciones (punteros), en total seis bytes, sin necesidad de ninguna otra operación. Es evidente que para este procedimiento el tiempo es totalmente independiente de la longitud de las dos cadenas, como nos muestran claramente los experimentos prácticos.

Por otra parte, la técnica que emplea el intérprete, aunque muy eficaz, puede producir serios inconvenientes cuando se haga una utilización intensiva de datos alfanuméricos en programas de una cierta complejidad. Procediendo de esta manera, en efecto, el BASIC provoca un enorme desperdicio de memoria, con la pretensión de obtener una velocidad muy superior, pero por desgracia nos encontramos con una auténtica rendición de cuentas: el tiempo que ha ahorrado hasta ese momento debe devolverlo de una vez y con intereses.

Pero procedamos con orden, intentando comprender más de cerca como actúa exactamente el BASIC con las variables alfanuméricas. Con referencia a la figura 1 del capítulo anterior, o sea, al mapa de la página cero del Commodore 64 (apéndice Q del libro de instrucciones), tecleemos el siguiente programa:

```
10 CLR
20 REM FINAL DE LA MEMORIA DISPONIBLE
30 TM=PEEK(55)+PEEK(56)*256
40 REM FINAL DEL AREA DE VARIABLES
50 REM (CRECE HACIA ARRIBA)
60 FA=PEEK(49)+PEEK(50)*256
70 REM FINAL DEL AREA DE CADENAS
80 REM (CRECE HACIA ABAJO)
90 BS=PEEK(51)+PEEK(52)*256
100 PRINT FA,BS,TM
110 PRINT "MEMORIA OCUPADA POR LAS CADENAS";
120 PRINT TM-BS
130 PRINT "MEMORIA TODAVIA DISPONIBLE";
140 PRINT BS-FA;" BYTES"
```

Poniendo en marcha el programa obtendremos:

```
2400      40960      40960
MEMORIA OCUPADA POR LAS CADENAS 0 BYTES
MEMORIA TODAVIA DISPONIBLE 38560 BYTES
```

Es obvio que el área de las cadenas no contiene nada, desde el momento en que ninguna variable alfanumérica ha sido definida.

Inserte ahora la línea siguiente, que introduce tres variables de cadena, pero todas iguales a la cadena vacía (a menudo llamada "nil"):

```
15 A$="": B$="": C$=""
```

Esta vez, el programa así modificado, nos da:

```
2447      40960      40960
MEMORIA OCUPADA POR LAS CADENAS 0 BYTES
MEMORIA TODAVIA DISPONIBLE 38517 BYTES
```

La disminución de la RAM disponible hay que imputarla, en parte, al alargamiento del programa, debido a la inserción de la nueva línea 15 y, en parte, al espacio requerido para memorización de las variables.

Ahora procedamos a atribuir un valor diferente de "nil" a las tres cadenas, modificando como sigue la línea 15:

```
15 A$="PRIMERA": B$="SEGUNDA": C$="TERCERA"
```


El programa señalará ahora el primer valor igual a 2460, mientras que la memoria todavía disponible ahora es igual a 38455; la pareja de valores 40960 y 40960, extrañamente, permanece invariable y sigue siendo nula la memoria ocupada por las cadenas. Aparentemente, los tres valores de cadena introducidos no aparecen por ninguna parte. La clave del enigma se desvela pronto; una vez más, el BASIC se muestra muy astuto; aprovecha que los contenidos de las variables alfanuméricas están contenidos en el mismo programa BASIC (que, recordamos a los desmemoriados, reside también en memoria). Así, los punteros de las tres variables A$, B$ y C$, se limitan a apuntar a estas situaciones del programa —aquellas en las que están materialmente escritos los valores "PRIMERA", "SEGUNDA" y "TERCERA", de nuestra línea 15— y no se requiere ningún espacio extra. Pero ¿qué ocurre si, manipulamos cadenas para crear otras nuevas cuyo contenido no se ha definido a priori? La respuesta es obvia: el intérprete no puede evitar reservar cierto espacio de memoria para este menester. Para convencernos experimentalmente añadamos esta otra línea:

```
25 C$=A$+B$+C$
```

y la respuesta será:

```
2476      40931      40960
MEMORIA OCUPADA POR LAS CADENAS 29 BYTES
MEMORIA TODAVIA DISPONIBLE 38455 BYTES
```

Esta vez, el intérprete ha tenido que utilizar, el depósito de las cadenas para colocar "PRIMERASEGUNDATERCERA", que constituye el nuevo contenido de C$. Por otra parte, el valor 29 señalado parece exagerado respecto a los 17 bytes que constituyen la nueva longitud de C$. También este misterio tiene explicación. La concatenación (suma) de tres cadenas no se hace de una sola vez, sino en dos fases: en la primera se suman las dos cadenas A$ y B$ obteniendo una cadena intermedia "PRIMERASEGUNDA"; en la segunda fase se concatena aquella a C$ para obtener su nuevo valor. El inconveniente reside en el hecho de que las cadenas auxiliares, utilizadas en las concatenaciones intermedias, no son eliminadas (para evitar complicaciones y ganar tiempo). Estas cadenas son como cadáveres que estorban, sin ningún fin, en la zona de las cadenas, sin estar "apuntadas", y además, por ninguna variable alfanumérica. Con nuestro programa, de momento, ciertamente no hay problemas, al disponer de 38455 bytes aún libres, pero no se necesita mucho para entender que, prosiguiendo tan alegremente, bien pronto hasta la vasta memoria de C64

se queda corta al llenarse de la "basura" (en inglés "garbage") dejada por las manipulaciones de datos alfanuméricos.

Un amontonamiento catastrófico de los resultados intermedios de numerosas operaciones no es tan raro como se podría creer: piense por ejemplo en la construcción de una cadena, carácter tras carácter, mediante sucesivas instrucciones GET y comprenderá la seriedad del problema, incluso en situaciones a primera vista sencillas. Una cadena de 250 caracteres, construida según el sistema que acabamos de mencionar, crearía 249 cadenas intermedias, de una longitud variable entre uno y 249 bytes. Haciendo cuentas se trata de un basurero inmenso: (249×248)/2, casi igual a ¡30 kilobytes! Nos podemos divertir un poco imaginando una situación parecida en el siguiente programa, que trata de valorar los efectos (perversos) del fenómeno en relación con la velocidad:

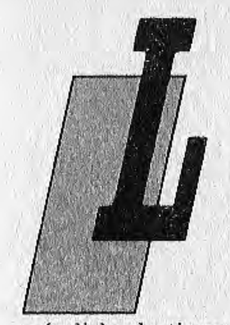
```
 1 CLR
10 V$=" ":J=0:K=0:N=0:TD=0:DIM A$(100)
20 FOR J=1 TO 100
30 V$=" ":TD=TI
40 FOR K=0 TO 250
50 V$=V$="A"
60 NEXT K
70 PRINT TI-TD,J
80 A$(J)=V$
90 NEXT J
```

Respiremos profundamente y dejemos para el próximo capítulo este y otros problemas.

CAPITULO VII

¿QUE HACER PARA QUE EL CARACOL CORRA?

¡Cronómetros listos!



ancémonos sin temor hacia el primer problema: dado que tanto las variables simples como las matrices se memorizan justo después de acabar el programa ¿qué sucedería si, en un momento dado de la ejecución, definiéramos una nueva variable? Está claro (por lo menos se puede intuir): el intérprete BASIC no tiene más remedio que desplazar todas las matrices (cada una con siete bytes); esta latosa operación supone una pérdida de tiempo que puede resultar notable si hay muchas matrices.

Con el programa siguiente es fácil verlo, midiendo experimentalmente la ralentización con el cronómetro interno del C64:

```
 1 CLR:REM RESET PUNTEROS
10 TD=0:REM PREDEFINE VARIABLE DE TRABAJO
20 DIM AA(1000):REM DEFINICION DE LA MATRIZ
30 TD=TI:REM DA LA SALIDA AL CRONOMETRO
40 A=0:B=0:C=0:REM DEFINE OTRAS TRES VARIABLES
50 PRINT TI-TD:REM TIEMPO TRANSCURRIDO
```

Resultado: 98×1/60 segundos (más de un segundo y medio) para desplazar todos los 3001 elementos de siete bytes hacia arriba; indispensable operación repetida tres veces, tantas cuantas variables (A, B y C) son introducidas. En otras palabras, con la inocente instrucción intermedia de la línea 40, se ha perdido un tiem-

po precioso sin ningún sentido práctico. Seguramente será instructivo realizar la medida efectuada en el programa anterior para distintos tipos y dimensiones de matrices. Nos daremos así cuenta que cuando se acusa al BASIC de ser un caracol hay bastante razón. Por ello, para que no sea algo temible su lentitud, debemos tomar medidas para, por lo menos, las situaciones más gravosas de pérdida improductiva de tiempo.

El caso que estamos examinando nos enseña claramente lo oportuno que resulta que en un programa se definan *al principio* todas las variables en juego, evitando cuidadosamente "golpes de ingenio" durante las instrucciones siguientes. El precio que se paga por no seguir esta regla de oro es el de temibles ralentizaciones, en un lenguaje ya de por sí no muy brillante en este aspecto. Además, siguiendo la sencilla regla que acabamos de dar, se matan dos pájaros de un tiro, ya que se obtiene también una mayor claridad en la documentación. Por ejemplo, en un programa de una cierta complejidad se recomienda que en las líneas iniciales esté contenida una serie de "Informaciones sobre los Datos" que resuman la organización. Nos podemos ayudar eficazmente con las REM, como en el ejemplo (indicativo) siguiente:

```
10 REM ***   RESUMEN DE DATOS   ***
20 REM *** 1-VALORES CONSTANTES ***
30 REM VERDADERO=-1:FALSO=0:RAIZ2=1.4142135
........
60 REM *** 2-DATOS DE ENTRADA ***
70 QT=0:PREC=0:NOM$=.....
........
100 REM *** 3-AREAS DE TRABAJO ***
110 IMP=0:SW=0:DEP=0:DEP$=".....
120 DIM VET(100),TAB$(200)....
...................
```

(en ninguno de los ejemplos de los capítulos 1 al 5 se ha adoptado este criterio, dada la sencillez de los casos examinados; esto no es óbice para que en algunos de estos casos elementales valga la pena introducirlo, para evitar los comportamientos lentísimos). Pero sigamos con el cronometraje en este otro programa:

```
1 CLR
10 K=0:TD=TI:REM DA SALIDA AL CRONOMETRO
20 FOR K=0 TO 3000
30 NEXT K
40 PRINT TI-TD:REM TIEMPO TRANSCURRIDO
```


El tiempo de ejecución resultado es de 240 "jiffys" (este extraño nombre corresponde a 1/60 de segundo), es decir, unos 4 segundos. Intente ahora sustituir en la línea 30 NEXT K por un simple NEXT: habrá una reducción a 200 jiffys, 43 menos. Esta no desdeñable reducción se debe al hecho de que si se especifica la variable K del ciclo FOR, el intérprete está obligado a comprobar, cada vez, su existencia en el área específica. Resultado: una inútil y perjudicial falta de tiempo, bastante impensable. Por lo tanto conviene no especificar en el NEXT la variable del bucle excepto cuando se corra el riesgo de dar lugar a alguna ambigüedad, como en los bucles anidados. Escribiendo el bucle todo entero en una misma línea el beneficio de velocidad obtenido es, en cambio, más modesto de lo que se podría pensar: exactamente 193 jiffys contra los 200 y 243 de los dos casos que hemos visto.

Probemos ahora con lo siguiente:

```
1 CLR
10 N=0:K=0:TD=TI
20 FOR K=0 TO 3000
30 N=N+1
40 PRINT TI-TD
```

El tiempo esta vez, es de 731 sesentavos de segundo. Si intentamos cambiar el tipo de operación tendremos:

30 N=N-1	tiempo=737 jiffys
30 N=N*1	tiempo=681 jiffys
30 N=N/1	tiempo=688 jiffys

Como vemos las diferencias son muy exiguas aunque, curiosamente, la multiplicación resulta la operación más rápida. Evidentemente, estos son los misterios de las operaciones en coma flotante. Es conveniente asimismo hacer las mismas pruebas con un número distinto a la unidad. Sustituyendo en los casos que acabamos de ver, el 1 por 1234, los tiempos resultarán iguales a:

1368 para la suma;
1372 para la resta;
1315 para la multiplicación;
1322 para la división.

Es decir: los tiempos aumentan considerablemente, casi se duplican, cuando se sale de los casos particulares como 1 ó 0. Llegados aquí, hagamos una modificación aparentemente inocente:

```
1 CLR
10 N=0:V=1234:TD=TI
20 FOR K=0 TO 3000
30 N=N+V
40 PRINT TI-TD
```

Los tiempos, para las cuatro operaciones aritméticas, resultarán ahora, por el mismo orden de antes: 662, 666, 609 y 616 jiffys, es decir, la mitad que antes, para hacer lo mismo. La explicación de este extraño hecho reside en que el intérprete sólo puede trabajar con variables del mismo formato (coma flotante con coma flotante, enteros con enteros) por lo que está obligado a hacer la traducción de uno de los dos operandos al formato del otro, trabajo bastante costoso en términos de tiempo (las diferencias pueden resultar trágicas cuando las instrucciones que conciernen a tipos mixtos se encuentran dentro de un bucle formado por muchísimas iteraciones).

Naturalmente, se podría obtener el mismo resultado sustituyendo en la línea 30 el valor 1 por 1234.0, pero es más práctico y elegante definir desde el principio el valor V de la constante. Haciéndolo así, sólo se perderá el tiempo necesario para la asignación al principio y de una vez por todas. La segunda recomendación, que en esencia va emparejada a la dada anteriormente, se podría enunciar así: es necesario evitar, hasta donde sea posible, los valores inmediatos, sustituyéndolos por las variables adecuadas, inicializadas al comienzo del programa de una vez por todas, para llamarlas cuando sean necesarias.

A estas variables que, en esencia, corresponden a constantes se les deberá dar, si es posible, nombres mnemónicos adecuados, tipo R2, PI (por "raíz de 2", número "pi") y similares.

Ahora complicaremos un poco más la vida al intérprete BASIC, añadiendo un bucle en un programa más largo en el que haya muchas variables activas en el momento de la ejecución. Añadamos al programa estas dos líneas:

```
2 A=33:B$="ABCD":C%=456
3 D=0:E=98765:F=0:G=0:H=0:J=75
```

Volvamos a ejecutar el programa, en apariencia inalterado en su parte central, de la que se mide la duración. Sin embargo, extrañamente, el tiempo de ejecución es de 848 sesentavos de segundo, un exceso de más de un tercio. Esta paradoja se explica según lo que hemos visto anteriormente sobre la técnica usada por el BASIC para memorizar las variables, es decir, cómo las pone una tras otra siguiendo el orden temporal de definición. Esta vez

"N" y "V" serán colocadas, respectivamente, en el décimo y undécimo lugar del área destinada a las variables no indexadas. El intérprete, todas las veces que tiene que acceder a ellas, tendrá que buscar entre los nombres, partiendo siempre desde el primero, hasta encontrar el que desea. La ralentización de la ejecución puede llegar a ser dramática en programas de cierta complejidad, en los que sea notable la cantidad de datos en juego. He aquí, otra recomendación práctica: resulta aconsejable cuidar el orden de definición de las variables de forma que, hasta donde sea posible, se pongan en los primeros puestos las de uso más corriente o las que necesiten de un acceso más rápido.

¿Y qué ocurre con los comentarios?

Alguno de ustedes podría llegar a temer, según lo visto, que los inconvenientes de la ralentización acechan a cada paso que den. Pero las sorpresas no sólo surgen al principio. Intenten añadir la siguiente línea:

```
25 REM COMENTARIO INUTIL
```

en el interior del bucle que estamos examinando: su duración pasará a 961 jiffys. ¿Cómo es posible que una instrucción que no comporta ningún tipo de elaboración haga perder tanto tiempo (113 jiffys de más)? Es probable que los más adelantados sepan encontrar una explicación exacta a esta pregunta. Será suficiente con que piensen en lo dicho sobre el comportamiento del intérprete, podríamos decir incluso sobre su propia naturaleza.

Les dejamos un poco de tiempo para pensar sobre esto y, mientras tanto, modificaremos la línea 25, alargando un poco el texto del comentario (por ejemplo: COMENTARIO QUE QUIZA SE PODRIA ELIMINAR); ahora el tiempo llegará a 1052 jiffys, lo que nos lleva a la conclusión de que la ralentización originada por un REM es también proporcional a su longitud.

¿Han encontrado la explicación que les pedimos? Probablemente se parecerá a la que vamos a dar: la naturaleza secuencial de la interpretación, es decir, su ejecución línea tras línea, supone también una notoria pérdida de tiempo en el examen de las instrucciones que, como los comentarios, no dan lugar a un proceso de traducción a lenguaje máquina ni, mucho menos, a la ejecución de un código. De todas formas, el intérprete no puede evitar hacer tal examen cada vez que, por así decirlo, se tropieza de nuevo con una REM. La reflexión que acabamos de hacer nos da pie para dar otro consejo: evite usar comentarios en el interior de los

bucles, y limítelos en todos los casos a la longitud mínima indispensable.

Seguramente alguno de ustedes dirá que se trata de un principio archiconocido; mejor así. Solamente queremos añadir que, cuando no se quiera renunciar a una utilización abundante de comentarios, lo mejor que se puede hacer es tener en la memoria de masa dos versiones del programa: la primera, sin REM, servirá para la ejecución, mientras la otra, rica en comentarios incluso kilométricos, se habilitará como copia de archivo para ser utilizada cada vez que se quiera trabajar sobre las distintas instrucciones para revisarlas o ponerlas al día.

Ahora, divirtámonos con este programa:

```
1 CLR
10 K=0:TD=TI
20 FOR K=0 TO 3000
30 GOSUB 1000
40 NEXT K
50 PRINT TD-TI
60 END
1000 RETURN
```

Tiempo: 504 jiffys. Si se sustituye el número 1000 de la subrutina por 100 se obtiene una pequeña ventaja (484 jiffys) debida a que al convertir en binario, de uso interno, un número a otro más pequeño requiere para su manejo menos tiempo. Pero no es esto todo. Añadamos instrucciones que, a primera vista, no influyan en la duración del bucle que estamos diseccionando, o sea:

```
1 CLR:GOTO 10
2 REM
3 REM
4 REM
5 REM
6 REM
7 REM
8 REM
9 REM
70 REM
71 REM
72 REM
73 REM
74 REM
```



```
75 REM
76 REM
77 REM
78 REM
79 REM
```

Esta vez la duración ha pasado a 611 jiffys, más del 25% de aumento. ¿Cómo es posible? Para entenderlo mejor realicemos una nueva modificación:

```
2 RETURN
30 GOSUB 2
```

obtendremos que la duración se reduce a 405 jiffys. El misterio parece hacerse más denso, pero pronto se aclarará si piensa en cómo actúa el intérprete en la búsqueda de un número de línea. Dado que las instrucciones están memorizadas una tras otra, en orden ascendente de línea, pero sin limitaciones sobre el paso de la numeración, lo único que puede hacer el BASIC es explorar todas, recorriéndolas a partir de la primera, hasta encontrar la que tenga el número indicado en el GOTO o GOSUB. Esto explica fácilmente cómo, para alcanzar la línea 1000, se utiliza más tiempo que para encontrar la línea 2. Conclusión: contrariamente a lo que suele hacerse es preferible, para conseguir una mayor velocidad, poner las subrutinas a la cabeza en lugar de a la cola del programa, como se hace en Pascal.

En el caso de programas muy largos la ventaja conseguida con tal proceder llega a ser enorme y capaz de reducir drásticamente los tiempos de respuesta. Evidentemente, la consideración anterior vale también para los GOTO y los THEN, aunque con ellos puede resultar difícil una intervención sin arriesgarse a alterar el flujo lógico correcto de las instrucciones. En cualquier caso, hay que intentar poner al principio las instrucciones que se ejecutan más a menudo.

Resumiendo, las reglas prácticas para aumentar la velocidad de un programa BASIC interpretado que hemos visto hasta ahora son las siguientes:

- escribir NEXT en lugar de NEXT y la variable del bucle;
- definir todas las variables y los vectores en las líneas iniciales, incluidas las constantes, a las que se les atribuirá un nombre simbólico;
- anteponer en el orden de definición los datos más utilizados y/o aquellos para los que sea necesario un acceso lo más rápido posible;

- evitar, o por lo menos limitar, el número y longitud de los REM;
- situar las subrutinas a la cabeza, y no a la cola, del programa.

Sería un útil ejercicio, una vez llegados aquí, repasar los ejemplos de los primeros capítulos, asociándolos con estos consejos prácticos.

De nuevo las cadenas

Seguramente las cadenas despertarán también el interés de los lectores, ya sea por lo curioso de su comportamiento, ya por la importancia de las ralentizaciones que pueden provocar, de las que es indispensable conocer sus causas y posibles remedios.

El tiempo invertido en la construcción de una cadena es de 91 jiffys la primera vez; después va creciendo, pero no de un modo uniforme, alcanzando el valor 102 en el paso 15 y bajando a 91 en el 16, para seguir creciendo continuamente hasta un valor de 156 en la cadena 43. Probando con DIM A$(500) todos los tiempos resultan alargados notablemente, alcanzando en el paso 49 un valor de 297 sesentavos, más del triple de lo que se empleó al principio. Evidentemente, cuando se da cuenta el intérprete de que no tiene más espacio a su disposición se ve obligado a eliminar las cadenas inútiles, reordenando las activas para poder continuar la ejecución de manera apropiada, lo que comporta una enorme cantidad de trabajo, ya que hay que compactar casi 40 kilobytes de memoria. Pero ¿cómo se explica que con un número de variables más elevado, 500 en lugar de 100, el tiempo resulte más largo? El número de cadenas creadas no tendría nada que ver con la extensión de la matriz. Esto se explica porque a mayor número de variables definidas más costosa resulta la primera parte (construcción) en términos de tiempo.

El proceso descrito anteriormente es llamado en inglés "garbage collection" (detección y eliminación de la información inservible) y es una expresión que explica elocuentemente el objeto de la reordenación del área de las cadenas. La operación supone grandes problemas en la ejecución de algunos programas complejos, ya sea porque puede dar lugar a ralentizaciones de bastantes décimas de segundo, ya porque la acción de limpieza puede empezar en momentos bastante imprevisibles que, si no se tiene suerte, podrían coincidir con los más críticos y menos indicados. Durante la recogida de la información inservible el ordenador aparece bloqueado y no responde a ninguna orden, incluida STOP, para volver a funcionar al final de la operación como si nada hubiera sucedido; un comportamiento engañoso pero plenamente justificado por la exigencia de poner remedio a una situación insostenible. Por lo tanto, no hay tiempo para gestionar otros recursos de E/S.

Quizá el único remedio practicable es el de prever la recogida de información inservible, forzando esta operación en un momento elegido cautelosamente por el programador. Esto se puede hacer recurriendo a la instrucción FRE(0). En efecto, la pregunta referente a la cantidad de memoria aún disponible obliga al BASIC a ordenar el espacio de las cadenas para dar una respuesta inexacta sobre las disponibilidades reales de espacio, lo cual puede aprovecharse adecuadamente. En el caso del programa que estamos examinando podemos añadir la línea:

```
22 K=FRE(0)
```

La duración del proceso de construcción de la cadena permanecerá constante (91) hasta el ciclo 23, pues la operación partirá siempre con la memoria reordenada y con todo el espacio sobrante a su disposición. Desgraciadamente, la presencia de variables activas reduce sucesivamente esta disponibilidad y, a partir del paso 24, se hace necesaria una recogida de información inservible incluso durante la creación de la simple cadena, lo que alarga el tiempo de ejecución. Por otra parte, la duración de la reordenación crece proporcionalmente al número de variables que se deban examinar y en este punto el BASIC entra en crisis.

Dos añadidos posteriores permiten analizar con precisión el funcionamiento de los tiempos de trabajo y de reordenación:

```
21 TD=TI
23 PRINT TI-TD
```

Paradójicamente es indispensable tener una gran capacidad de memoria para conseguir una cierta velocidad en el tratamiento de las cadenas. No es tanto una cuestión de espacio que pueda ser ocupado, como la consecuencia indirecta de la cantidad de veces que tiene que realizarse necesariamente la recogida de información inservible: cuanto más grande sea la capacidad del área de las cadenas menos veces tendrá que intervenir el ordenador para hacer una reordenación.

Es una buena regla limitar al mínimo el número de las variables, es decir, abreviar la primera fase de reordenación; lamentablemente no se puede hacer mucho para superar este grave e insospechado límite, típico de los lenguajes interpretados en general.

Añadiéndole a nuestro programa las líneas siguientes:

```
51 D=PEEK(51)+PEEK(52)*256+(PEEK(49)+PEEK(50)*256)
52 PRINT D,K
```

será posible hacerse una idea sobre ocupación de la memoria durante la complicada construcción de una larga cadena.

Nos referiremos, por último, a otra consecuencia de la "Recogida de Información Inservible". La presencia en el C64 de una puerta RS232 es un instrumento sencillo y económico para la comunicación, también a gran distancia, entre ordenadores. Quien lo haya probado se habrá dado cuenta que las cosas en BASIC no siempre marchan bien. Aunque no se tienen problemas para la salida de datos, incluso a grandes velocidades (2.400 bits por segundo), es mucho más difícil intentar adquirir datos, especialmente con la instrucción GET. En efecto, muy pronto, al recurrir a reordenaciones de memoria (que, como hemos visto, inhiben las operaciones de E/S) no se permite al BASIC descargar el almacenamiento intermedio del canal serie al mismo ritmo con el que se carga el Sistema Operativo, por lo que el ordenador entrará en crisis obstinadamente. Disminuir la velocidad del canal RS232 sólo sirve para aplazar el bloqueo del sistema, dado que la duración de la recogida de información inservible tiende a aumentar con el crecimiento de las variables activas y a hacerse más frecuente según la memoria disponible se reduce. La única forma de superar las limitaciones impuestas por el lenguaje interpretado viene dada por la adopción de rutinas de manejo escritas en lenguaje máquina.

Compiladores de BASIC, el punto y final

Es éste un tema a menudo motivo de encendidas discusiones entre los usuarios de pequeños sistemas. Se trata de programas traductores capaces de "preinterpretar" el programa fuente traduciéndolo a lenguaje evolucionado, y ganando así tiempo en el momento de la ejecución del programa objeto (o sea, el programa fuente traducido al lenguaje máquina).

Hay quienes piensan que estos aparatos transforman un programa fuente en BASIC a un programa en lenguaje máquina idéntico (o casi) a aquel que hubiera sido escrito por un programador mediano en lenguaje Ensamblador. Contrariamente a lo que asegura la publicidad esto está muy lejos de ser cierto. Los compiladores más corrientes se limitan, casi siempre, a traducir únicamente las instrucciones de BASIC a los correspondientes saltos a sus respectivas subrutinas de manejo. Evidentemente, se obtiene un cierto aumento de la velocidad ligado al hecho de que se elimina el tiempo de interpretación, pero por otra parte la mayoría de estos compiladores (hablamos de los que se utilizan para ordenadores no profesionales) hacen muy poco más, limitándose a sencillas modificaciones, no tanto para optimizar el código sino para aportar pequeños retoques como la eliminación de todos los comentarios.

Según nuestra experiencia personal hay que estar prevenidos frente a ciertos compiladores: las ventajas que se puedan obtener (mediocres muchas veces) pueden no compensar los gastos y disgustos con que nos podemos encontrar en la compilación. En cualquier caso, sólo un programa BASIC que esté bien optimizado podrá beneficiarse de las ventajas prometidas por la compilación, y siempre que el tiempo de interpretación de las instrucciones haya llegado a un porcentaje no despreciable con respecto al de ejecución.

Concluyendo: para dar un buen empuje al caracol BASIC es una buena regla seguir los consejos prácticos que les hemos recomendado sin olvidar, naturalmente, el planteamiento general: hacer un programa correctamente es difícil, pero posible.

BIBLIOGRAFIA

Método general de análisis de una aplicación informática. (Tomo 2: etapas y puntos fundamentales del análisis orgánico y de la programación.)
X. Castellani. *Masson.*

Introducción a la programación. (Tomo 1: algorítmica y lenguajes.)
Brondi y Clavel. *Masson.*

Matemáticas para la informática personal.
Lehning y Jakubowicz. *Masson.*

Programación avanzada del Amstrad.
Don Thomasson. *Anaya Multimedia.*

Programación en BASIC: un método práctico.
Dachslager y Zucker. *Anaya Multimedia.*

Diseño de gráficos y videojuegos. Tratamiento en tres dimensiones.
Angel y Jones, 1985. *Anaya Multimedia.*

Programación avanzada en BASIC.
Bishop. *Anaya Multimedia.*

El libro del IBM, PC, XT, AT.
L. E. Frenzel Jr. / L. E. Frenzel III, 1985. *Anaya Multimedia.*

Diccionario de informática ingles-español-francés.
G. A. Mania, 1985. *Paraninfo.*

Diccionario del BASIC.
Willie Hart, 1985. *Paraninfo.*

Como programar su COMMODORE 64 1 - BASIC, gráficos, sonido.
F. Montell, 1985. *Paraninfo.*

Como programar un COMMODORE 64 2 - Lenguaje máquina, E/S, periféricos.
F. Montell, 1985. *Paraninfo.*

Tratamientos de textos con BASIC.
G. Quaneaux, 1985. *Paraninfo.*

Informática para no avanzados.
G. Willmott, 1985. *Deusto.*

102 programas para su APPLE.
Desconchat. *Elisa.*

Claves para el COMMODORE 64.
D. Gean David. *Elisa.*

COMMODORE 64, para todos.
Boisgontier Brebion Foucault. *Elisa.*

Los ordenadores. Fundamentos y sistemas.
J. C. Giarratano, 1984. *Díaz de Santos.*

Conceptos actuales sobre la tecnología de los ordenadores.
J. C. Giarratano, 1984. *Díaz de Santos.*

Diccionario de informática ingles-español. Glosario de términos informáticos.
Olivetti, 5. Edic, 1984. *Paraninfo.*

Como programar ordenadores personales.
R. Farrando, 1985. *Marcombo.*

Diccionario de informática.
Masson, 2. Edic, 1985. *Masson.*

Glosario de computación.
Alan Freedman, 1984. *McGraw-Hill.*

Diccionario del BASIC.
A. Lien. *Ed. Elisa.*

Claves para el Appe II.
Breaud-Pouliquen. *Ed. Elisa.*

Claves para el Commodore 64.
D. J. David. *Ed. Elisa.*

Claves para el ZX-Spectrum.
J. Francois Séhan. *Ed. Elisa.*

El BASIC de la A a la Z.
J. Boisgontrer. *Ed. Elisa.*

Commodore 64 para todos.
J. Boisgontrer. *Ed. Elisa.*

102 Programas para Commodore 64.
J. Deconchat. *Ed. Elisa.*

102 Programas para ZX81 y Spectrum.
J. Deconchat. *Ed. Elisa.*

El Apple y sus ficheros.
J. Boisgontrer. *Ed. Elisa.*

Microordenadores y cassettes.
M. Salem. *Ed. Noray.*

Profundizando en el ZX-Spectrum.
D. Jones. *Ed. Noray.*

BIBLIOTECA BASICA INFORMATICA

INDICE GENERAL

10 Sistemas operativos y software de base
Qué son, para qué sirven. Unos desconocidos muy importantes.

11 Sistema operativo CP/M
Uno de los sistemas operativos para microprocesadores de 8 bits de mayor difusión en el mercado.

12 MS-DOS: el estándar de IBM
Sistema operativo para el microprocesador de 16 bits 8088, adoptado por el IBM-PC.

13 Paquetes de aplicaciones. Software "pret a porter"
Características y peculiaridades de los más importantes paquetes de aplicaciones.

14 VisiCalc: una buena hoja de cálculo
Interioridades y manejo de una de las hojas de cálculo más usadas.

15 Dibujar con el ordenador
Profundizando en una de las facetas útiles y divertidas que nos ofrecen los ordenadores.

16 Tratamiento de textos... para escribir con el ordenador
Cómo convertir su ordenador en una máquina de escribir con memoria y todo tipo de posibilidades.

17 Diseño de juegos
Particularidades características de esta aplicación de los ordenadores.

18 LOGO: la tortuga inteligente
Un lenguaje conocido por su «cursor gráfico», la tortuga, y sus aplicaciones pedagógicas al alcance de su mano.

19 BASIC y tratamiento de imágenes
Todo lo que en **¡Y llegó el BASIC!** no se pudo ver sobre las imágenes y gráficos en el BASIC.

20 Bancos de datos (I)
Peculiaridades de una de las aplicaciones de los ordenadores más interesantes, y que más dinero mueven.

21 Bancos de datos (II)
Profundizando en sus características.

22 Paquetes integrados: Lotus 1-2-3 y Simphony
Estudio de dos de los paquetes integrados (Hoja de cálculo + base de datos + ...) más conocidos.

23 dBASE II y dBASE III
Cómo aprovechar las dos versiones más recientes de esta importante base de datos.

24 Los ordenadores uno a uno
Un amplio y completo estudio comparativo.

25 Cálculo numérico en BASIC
Una aplicación especializada a su disposición.

26 Multiplan
Cómo hacer uso de este moderno paquete de aplicaciones.

27 FORTRAN y COBOL
Dos lenguajes muy especializados y distintos.

28 FORTH: anatomía de un lenguaje inteligente
Principales características de un lenguaje moderno, flexible y de amplio uso, en la robótica.

29 Cómo realizar nuestro propio banco de datos
Conocimientos necesarios para poder fabricar un banco de datos a nuestro gusto y medida.

30 Los paquetes integrados uno a uno
Todos los que usted puede encontrar en el mercado.

NOTA: Ingelek, S. A. se reserva el derecho de modificar, sin previo aviso, el orden, título o contenido de cualquier volumen de la colección.

NOTAS



i deseamos que nuestros programas en BASIC sean flexibles, rápidos, y efectivos no basta con un mero conocimiento de las instrucciones a nuestra disposición y de la función de cada una de ellas. Debemos llegar un poco más lejos.

Los algoritmos son, por ejemplo, un punto clave. Tanto si se reducen simplemente a problemas numéricos como si exigen una línea de reflexión (una estrategia) debemos tener claro el modo de enfrentarnos a su programación. Asimismo es de gran importancia conocer la forma en la que el intérprete BASIC maneja la memoria, asignando, moviendo y borrando variables, constantes y matrices, y la manera de lograr que un lenguaje tan lento no lo sea más todavía sino que, por el contrario, gane en rapidez.

Todos estos temas, arropados con gran cantidad de ejemplos, serán el núcleo de este libro.

395 pts.
(incluido IVA)